# KNN classification

In this exercise you'll explore a subset of the [**Large Movie Review Dataset**](http://ai.stanford.edu/~amaas/data/sentiment/). The variables X\_train, X\_test, y\_train, and y\_test are already loaded into the environment. The X variables contain features based on the words in the movie reviews, and the y variables contain labels for whether the review sentiment is positive (+1) or negative (-1).

*This course touches on a lot of concepts you may have forgotten, so if you ever need a quick refresher, download the*[***Scikit-Learn Cheat Sheet***](https://datacamp-community-prod.s3.amazonaws.com/5433fa18-9f43-44cc-b228-74672efcd116)*and keep it handy!*

**Instructions**

**100 XP**

* Create a KNN model with default hyperparameters.
* Fit the model.
* Print out the prediction for the test example 0.

[**Take Hint (-30 XP)**](javascript:void(0))

from sklearn.neighbors import KNeighborsClassifier

# Create and fit the model

knn = KNeighborsClassifier()

knn.fit(X\_train, y\_train)

# Predict on the test features, print the results

pred = knn.predict(X\_test)[0]

print("Prediction for test example 0:", pred)

In [1]: help(knn.fit)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

help(knn.fit)

NameError: name 'knn' is not defined

In [2]: KNeighborsClassifier

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KNeighborsClassifier

NameError: name 'KNeighborsClassifier' is not defined

Traceback (most recent call last):

File "script.py", line 8, in <module>

pred = knn.predict(x\_test)[0]

NameError: name 'x\_test' is not defined

<script.py> output:

Prediction for test example 0: 1.0

In [3]:

 +100 XP

Nice work! Looks like you remember how to use scikit-learn for supervised learning.

**Comparing models**

Compare k nearest neighbors classifiers with k=1 and k=5 on the handwritten digits data set, which is already loaded into the variables X\_train, y\_train, X\_test, and y\_test. You can set k with the n\_neighbors parameter when creating the KNeighborsClassifier object, which is also already imported into the environment.

Which model has a higher test accuracy?

**Instructions**

**50 XP**

**Possible Answers**

k=1

k=5

>>> from sklearn.neighbors import KNeighborsClassifier as knc

>>> knn = knc()

>>> knc(5)

KNeighborsClassifier(algorithm='auto', leaf\_size=30, metric='minkowski',

metric\_params=None, n\_jobs=None, n\_neighbors=5, p=2,

weights='uniform')

>>> knc(1)

KNeighborsClassifier(algorithm='auto', leaf\_size=30, metric='minkowski',

metric\_params=None, n\_jobs=None, n\_neighbors=1, p=2,

weights='uniform')

+50 XP

Great! You've just done a bit of model selection!

**Overfitting**

Which of the following situations looks like an example of overfitting?

**Answer the question**

**50 XP**

**Possible Answers**

Training accuracy 50%, testing accuracy 50%.

Training accuracy 95%, testing accuracy 95%.

Training accuracy 95%, testing accuracy 50%.

Training accuracy 50%, testing accuracy 95%.

 +50 XP

Great job! Looks like you understand overfitting.

**Running LogisticRegression and SVC**

In this exercise, you'll apply logistic regression and a support vector machine to classify images of handwritten digits.

**Instructions**

**100 XP**

* Apply logistic regression and SVM (using SVC()) to the handwritten digits data set using the provided train/validation split.
* For each classifier, print out the training and validation accuracy.

[**Take Hint (-30 XP)**](javascript:void(0))

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))

 +100 XP

Nicely done! Later in the course we'll look at the similarities and differences of logistic regression vs. SVMs.

**Exercise**

**Exercise**

**Sentiment analysis for movie reviews**

In this exercise you'll explore the probabilities outputted by logistic regression on a subset of the [**Large Movie Review Dataset**](http://ai.stanford.edu/~amaas/data/sentiment/).

The variables X and y are already loaded into the environment. X contains features based on the number of times words appear in the movie reviews, and y contains labels for whether the review sentiment is positive (+1) or negative (-1).

**Instructions**

**100 XP**

* Train a logistic regression model on the movie review data.
* Predict the probabilities of negative vs. positive for the two given reviews.
* Feel free to write your own reviews and get probabilities for those too!

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Have you specified the arguments for lr.fit() using the right syntax?

# 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])

In [1]: X\_train

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

X\_train

NameError: name 'X\_train' is not defined

In [2]: X

...

Out[2]:

<2000x2500 sparse matrix of type '<class 'numpy.float64'>'

with 209727 stored elements in Compressed Sparse Row format>

In [3]: X

Out[3]:

<2000x2500 sparse matrix of type '<class 'numpy.float64'>'

with 209727 stored elements in Compressed Sparse Row format>

In [4]:

In [4]:

In [4]: In[5]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

In[5]

IndexError: list index out of range

In [5]: Out[5]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

Out[5]

KeyError: 5

In [6]: Out[5]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

Out[5]

KeyError: 5

Traceback (most recent call last):

File "script.py", line 3, in <module>

lr.fit(X)

TypeError: fit() missing 1 required positional argument: 'y'

<script.py> output:

Review: LOVED IT! This movie was amazing. Top 10 this year.

Probability of positive review: 0.8079007873616059

Review: Total junk! I'll never watch a film by that director again, no matter how good the reviews.

Probability of positive review: 0.5855117402793947

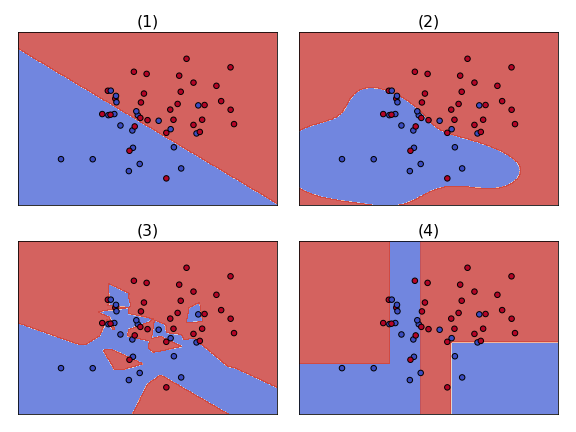
In [7]:

+100 XP

Fantastic! The second probability would have been even lower, but the word "good" trips it up a bit, since that's considered a "positive" word.

**Which decision boundary is linear?**

Which of the following is a linear decision boundary?



**Answer the question**

**50 XP**

**Possible Answers**

(1)

press

1

(2)

press

2

(3)

press

3

(4)

press

4

+50 XP

Good job! You correctly identified the linear decision boundary.

**Exercise**

**Exercise**

**Visualizing decision boundaries**

In this exercise, you'll visualize the decision boundaries of various classifier types.

A subset of scikit-learn's built-in wine dataset is already loaded into X, along with binary labels in y.

**Instructions**

**100 XP**

* Create the following classifier objects with default hyperparameters: LogisticRegression, LinearSVC, SVC, KNeighborsClassifier.
* Fit each of the classifiers on the provided data using a for loop.
* Call the plot\_4\_classifers() function (similar to the code [**here**](https://scikit-learn.org/stable/auto_examples/svm/plot_iris_svc.html)), passing in X, y, and a list containing the four classifiers.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Make sure you instantiated all the classifiers correctly. They should be in the same order as in the instructions.

[**Show Answer (-70 XP)**](javascript:void(0))

**Hint**

* Everything you need for the classifiers list is imported for you already.
* Don't forget to actually instantiate these with () after each classifier name, for example LogisticRegression() with the parenthesis. Do this for all of the classifiers.

 Awesome, thanks for your feedback!

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()

Traceback (most recent call last):

File "script.py", line 13, in <module>

plot\_4\_classifiers(X, y, classifiers)

File "script.py", line 98, in plot\_4\_classifiers

plot\_classifier(X, y, clf, ax, ticks=True)

File "script.py", line 64, in plot\_classifier

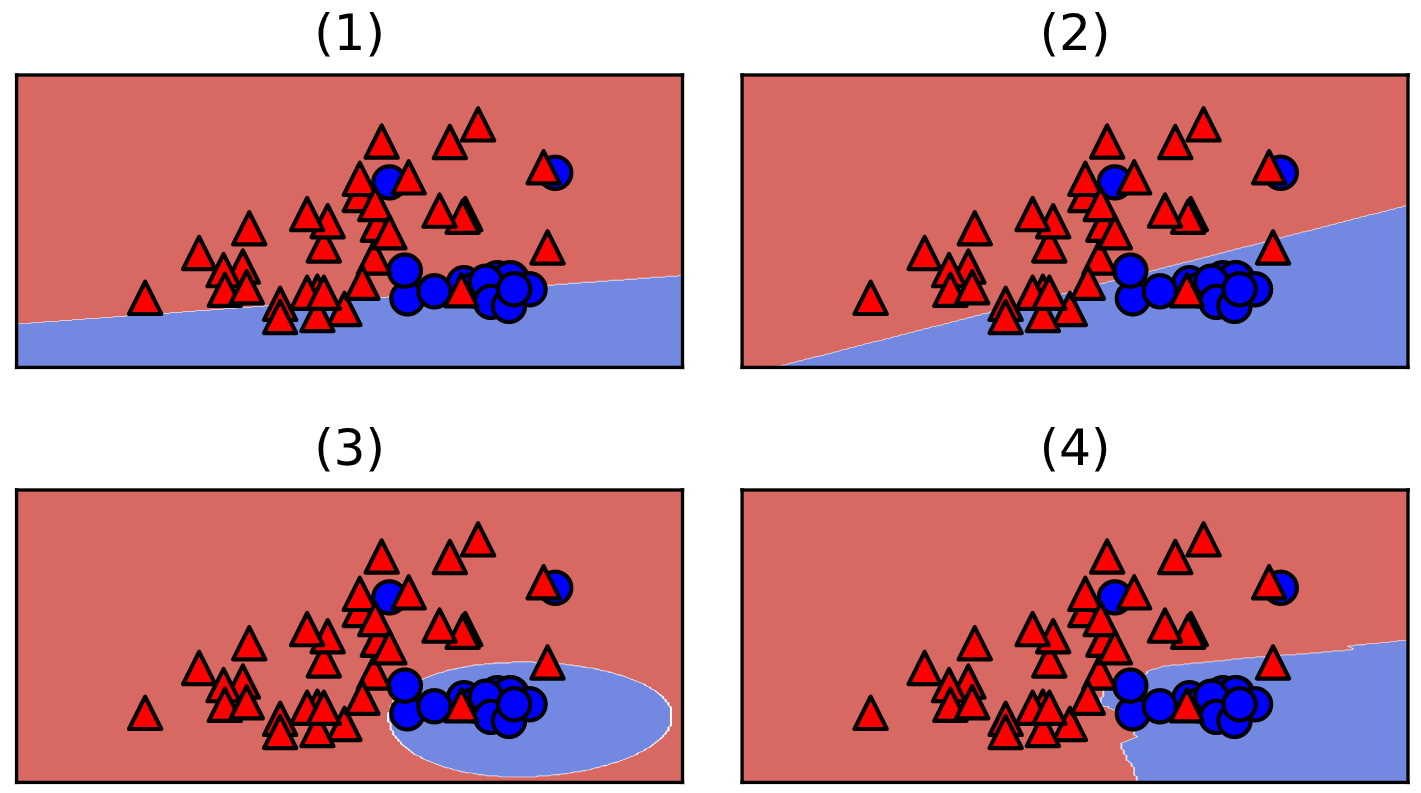
cs = plot\_contours(ax, clf, xx, yy, cmap=plt.cm.coolwarm, alpha=0.8, proba=proba)

File "script.py", line 46, in plot\_contours

Z = clf.predict(np.c\_[xx.ravel(), yy.ravel()])

TypeError: predict() missing 1 required positional argument: 'X'

In [1]:



+100 XP

Nice! As you can see, logistic regression and linear SVM are linear classifiers whereas the default SVM and KNN are not.

**How models make predictions**

Which classifiers make predictions based on the sign (positive or negative) of the raw model output?

**Answer the question**

**50 XP**

**Possible Answers**

Logistic regression only

press

1

Linear SVMs only

press

2

Neither

press

3

Both logistic regression and Linear SVMs

press

4

**Incorrect Submission**

Remember, logistic regression and linear SVMs make predictions in the same way.

+50 XP

Nice! Furthermore, since logistic regression and SVMs are both linear classifiers, the raw model output is a linear function of x.

**Exercise**

**Exercise**

**Changing the model coefficients**

When you call fit with scikit-learn, the logistic regression coefficients are automatically learned from your dataset. In this exercise you will explore how the decision boundary is represented by the coefficients. To do so, you will change the coefficients manually (instead of with fit), and visualize the resulting classifiers.

A 2D dataset is already loaded into the environment as X and y, along with a linear classifier object model.

**Instructions**

**100 XP**

* Set the two coefficients and the intercept to various values and observe the resulting decision boundaries.
* Try to build up a sense of how the coefficients relate to the decision boundary.
* Set the coefficients and intercept such that the model makes no errors on the given training data.

[**Take Hint (-30 XP)**](javascript:void(0))

# Set the coefficients

model.coef\_ = np.array([[0,1]])

model.intercept\_ = np.array([0])

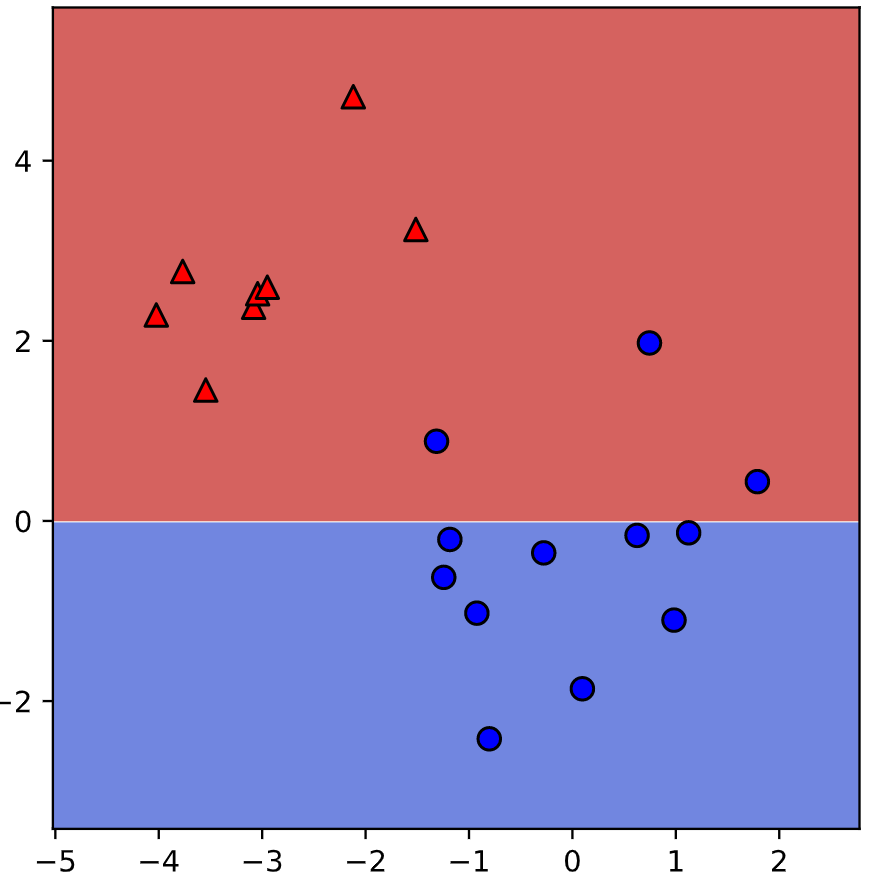
# Plot the data and decision boundary

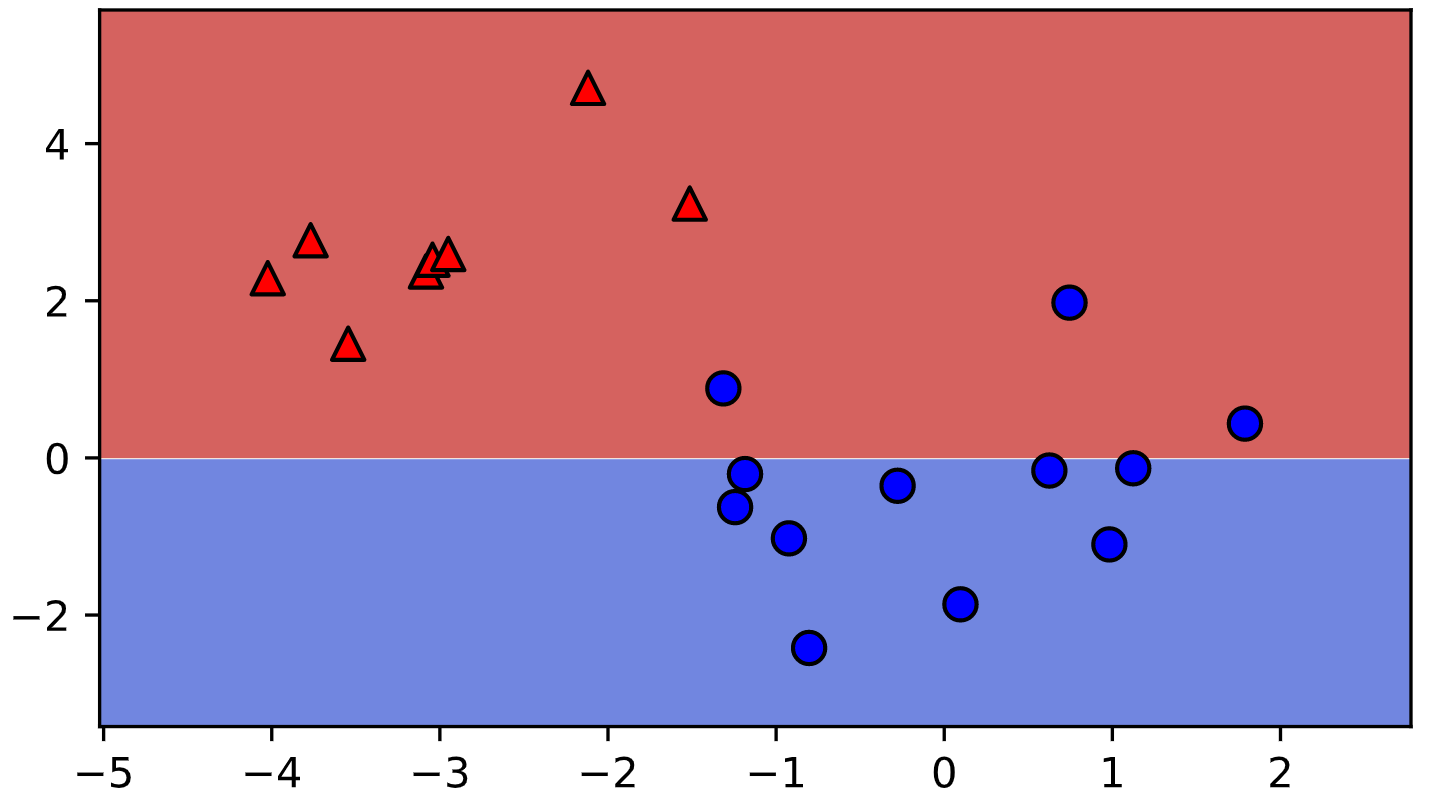
plot\_classifier(X,y,model)

# Print the number of errors

num\_err = np.sum(y != model.predict(X))

print("Number of errors:", num\_err)





<script.py> output:

Number of errors: 3

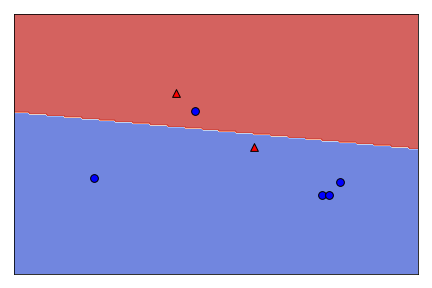
In [1]:

+100 XP

Great job! As you've been experiencing, the coefficients determine the slope of the boundary and the intercept shifts it.

**The 0-1 loss**

In the figure below, what is the 0-1 loss (number of classification errors) of the classifier?



**Answer the question**

**50 XP**

**Possible Answers**

0

1

2

3

**Incorrect Submission**

There is 1 misclassified red point and misclassified blue point.

+50 XP

Correct! There is 1 misclassified red point and 1 misclassified blue point.

**Exercise**

**Exercise**

**Minimizing a loss function**

In this exercise you'll implement linear regression "from scratch" using scipy.optimize.minimize.

We'll train a model on the Boston housing price data set, which is already loaded into the variables X and y. For simplicity, we won't include an intercept in our regression model.

**Instructions**

**100 XP**

* Fill in the loss function for least squares linear regression.
* Print out the coefficients from fitting sklearn's LinearRegression.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Make sure you correctly updated the value of s. That's all you should change in the function definition.

**Hint**

* The loss is the square of the difference between the true and predicted y-values (because we want them to be similar).
* The access the coefficients, use lr.coef\_.

# The squared error, summed over training examples

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\_pred-y\_i\_true)\*\*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\_)

Traceback (most recent call last):

File "script.py", line 12, in <module>

w\_fit = minimize(my\_loss, X[0]).x

File "script.py", line 604, in minimize

return \_minimize\_bfgs(fun, x0, args, jac, callback, \*\*options)

File "script.py", line 1009, in \_minimize\_bfgs

gfk = myfprime(x0)

File "script.py", line 327, in function\_wrapper

return function(\*(wrapper\_args + args))

File "script.py", line 765, in approx\_fprime

return \_approx\_fprime\_helper(xk, f, epsilon, args=args)

File "script.py", line 702, in \_approx\_fprime\_helper

raise ValueError("The user-provided "

ValueError: The user-provided objective function must return a scalar value.

<script.py> output:

[-9.16299112e-02 4.86754828e-02 -3.77698794e-03 2.85635998e+00

-2.88057050e+00 5.92521269e+00 -7.22470732e-03 -9.67992974e-01

1.70448714e-01 -9.38971600e-03 -3.92421893e-01 1.49830571e-02

-4.16973012e-01]

[-9.16297843e-02 4.86751203e-02 -3.77930006e-03 2.85636751e+00

-2.88077933e+00 5.92521432e+00 -7.22447929e-03 -9.67995240e-01

1.70443393e-01 -9.38925373e-03 -3.92425680e-01 1.49832102e-02

-4.16972624e-01]

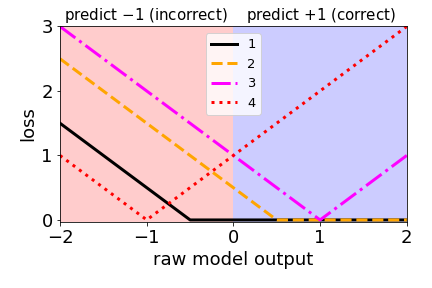
In [1]:

+70 XP

Great job! This was a tough one. Isn't it cool how you reproduce the weights learned by scikit-learn?

**Classification loss functions**

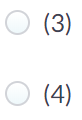
Which of the four loss functions makes sense for classification?



**Answer the question**

**50 XP**

**Possible Answers**



+50 XP

Correct! This loss is very similar to the hinge loss used in SVMs (just shifted slightly).

**Exercise**

**Exercise**

**Comparing the logistic and hinge losses**

In this exercise you'll create a plot of the logistic and hinge losses using their mathematical expressions, which are provided to you.

The loss function diagram from the video is shown on the right.

**Instructions**

**100 XP**

* Evaluate the log\_loss() and hinge\_loss() functions **at the grid points** so that they are plotted.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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

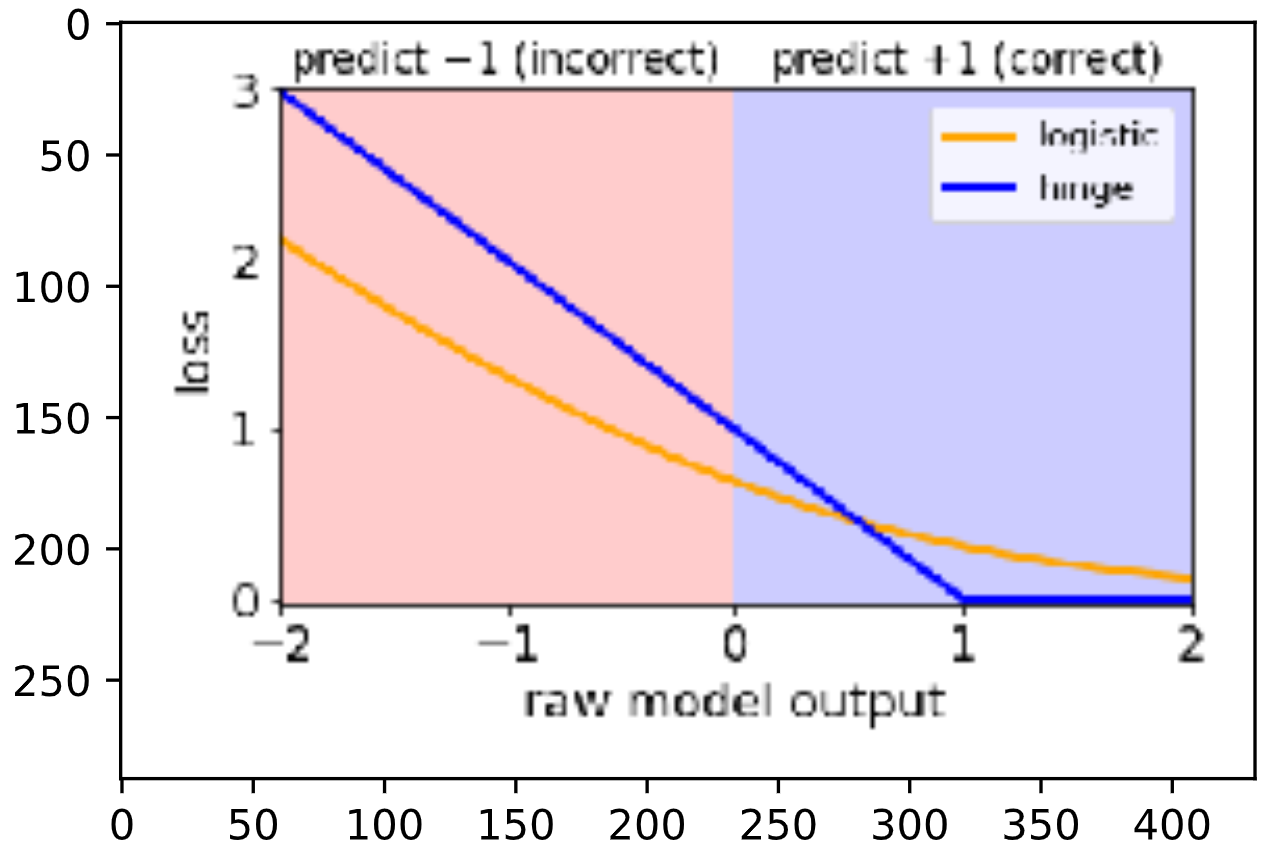
grid = np.linspace(-2, 2, 1000)

plt.plot(grid, log\_loss(grid), label='logistic')

plt.plot(grid, hinge\_loss(grid), label='hinge')

plt.legend()

plt.show()



In [1]: raw\_model\_output

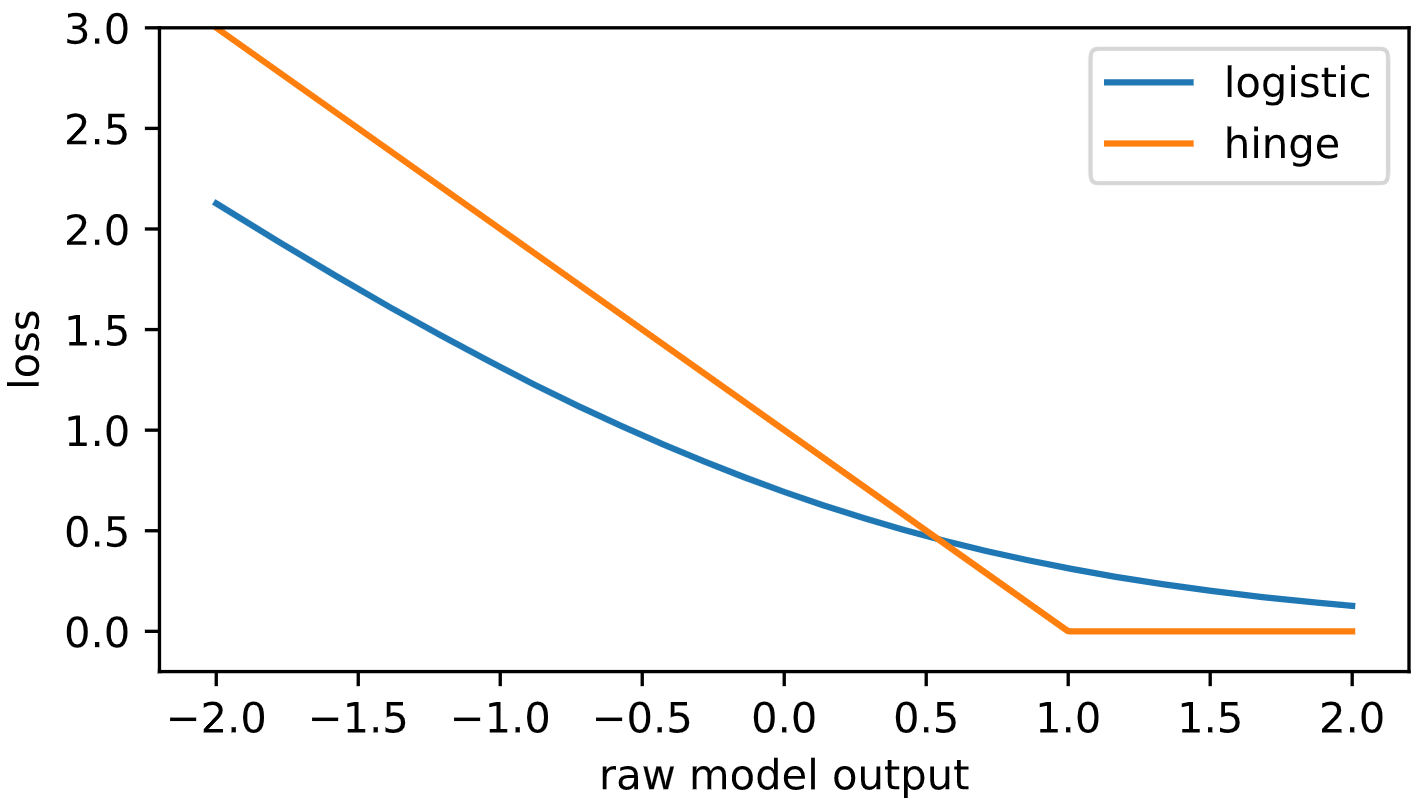
Traceback (most recent call last):

File "<stdin>", line 1, in <module>

raw\_model\_output

NameError: name 'raw\_model\_output' is not defined

In [2]:



+100 XP

Nice! As you can see, these match up with the loss function diagrams we saw in the video.

**Exercise**

**Exercise**

**Implementing logistic regression**

This is very similar to the earlier exercise where you implemented linear regression "from scratch" using scipy.optimize.minimize. However, this time we'll minimize the logistic loss and compare with scikit-learn's LogisticRegression (we've set C to a large value to disable regularization; more on this in Chapter 3!).

The log\_loss() function from the previous exercise is already defined in your environment, and the sklearn breast cancer prediction dataset (first 10 features, standardized) is loaded into the variables X and y.

**Instructions**

**100 XP**

* Input the number of training examples into range().
* Fill in the loss function for logistic regression.
* Compare the coefficients to sklearn's LogisticRegression.

[**Take Hint (-30 XP)**](javascript:void(0))

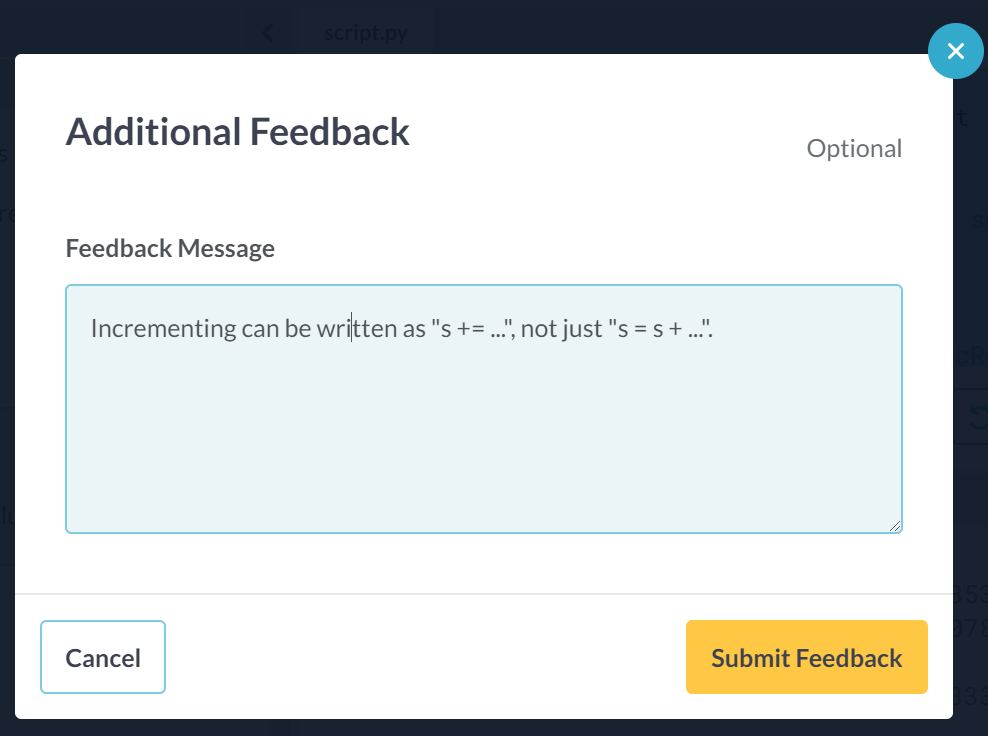
**Incorrect Submission**

Your code can not be executed due to a syntax error:  
invalid syntax (script.py, line 2).

Make sure you correctly updated the value of s.

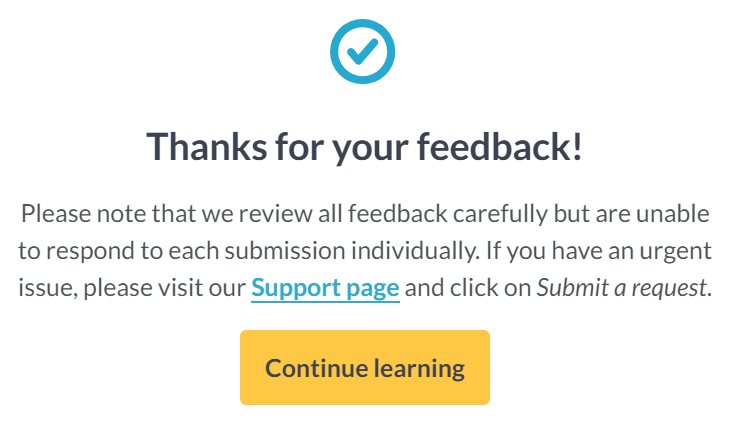
**Feedback Message**

Incrementing can be written as "s += ...", not just "s = s + ...".



**Thanks for your feedback!**

Please note that we review all feedback carefully but are unable to respond to each submission individually. If you have an urgent issue, please visit our [**Support page**](https://support.datacamp.com/) and click on *Submit a request*.



# The logistic loss, summed over training examples

def my\_loss(w):

s = 0

for i in range(y.size):

raw\_model\_output = w@X[i]

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\_)

File "script.py", line 2

def my\_loss(y.size):

^

SyntaxError: invalid syntax

<script.py> output:

[ 1.03592182 -1.65378492 4.08331342 -9.40923002 -1.06786489 0.07892114

-0.85110344 -2.44103305 -0.45285671 0.43353448]

[[ 1.03731085 -1.65339037 4.08143924 -9.40788356 -1.06757746 0.07895582

-0.85072003 -2.44079089 -0.45271 0.43334997]]

<script.py> output:

[ 1.03592182 -1.65378492 4.08331342 -9.40923002 -1.06786489 0.07892114

-0.85110344 -2.44103305 -0.45285671 0.43353448]

[[ 1.03731085 -1.65339037 4.08143924 -9.40788356 -1.06757746 0.07895582

-0.85072003 -2.44079089 -0.45271 0.43334997]]

In [1]:

+100 XP

Great job! As you can see, logistic regression is just minimizing the loss function we've been looking at. Much more on logistic regression in the next chapter!

**Exercise**

**Exercise**

**Regularized logistic regression**

In Chapter 1, you used logistic regression on the handwritten digits data set. Here, we'll explore the effect of L2 regularization.

The handwritten digits dataset is already loaded, split, and stored in the variables X\_train, y\_train, X\_valid, and y\_valid. The variables train\_errs and valid\_errs are already initialized as empty lists.

**Instructions**

**100 XP**

* Loop over the different values of C\_value, creating and fitting a LogisticRegression model each time.
* Save the error on the training set and the validation set for each model.
* Create a plot of the training and testing error as a function of the regularization parameter, C.
* Looking at the plot, what's the best value of C?

[**Take Hint (-30 XP)**](javascript:void(0))

**Hint**

* Create each LogisticRegression object using the C\_value from the loop.

# 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)

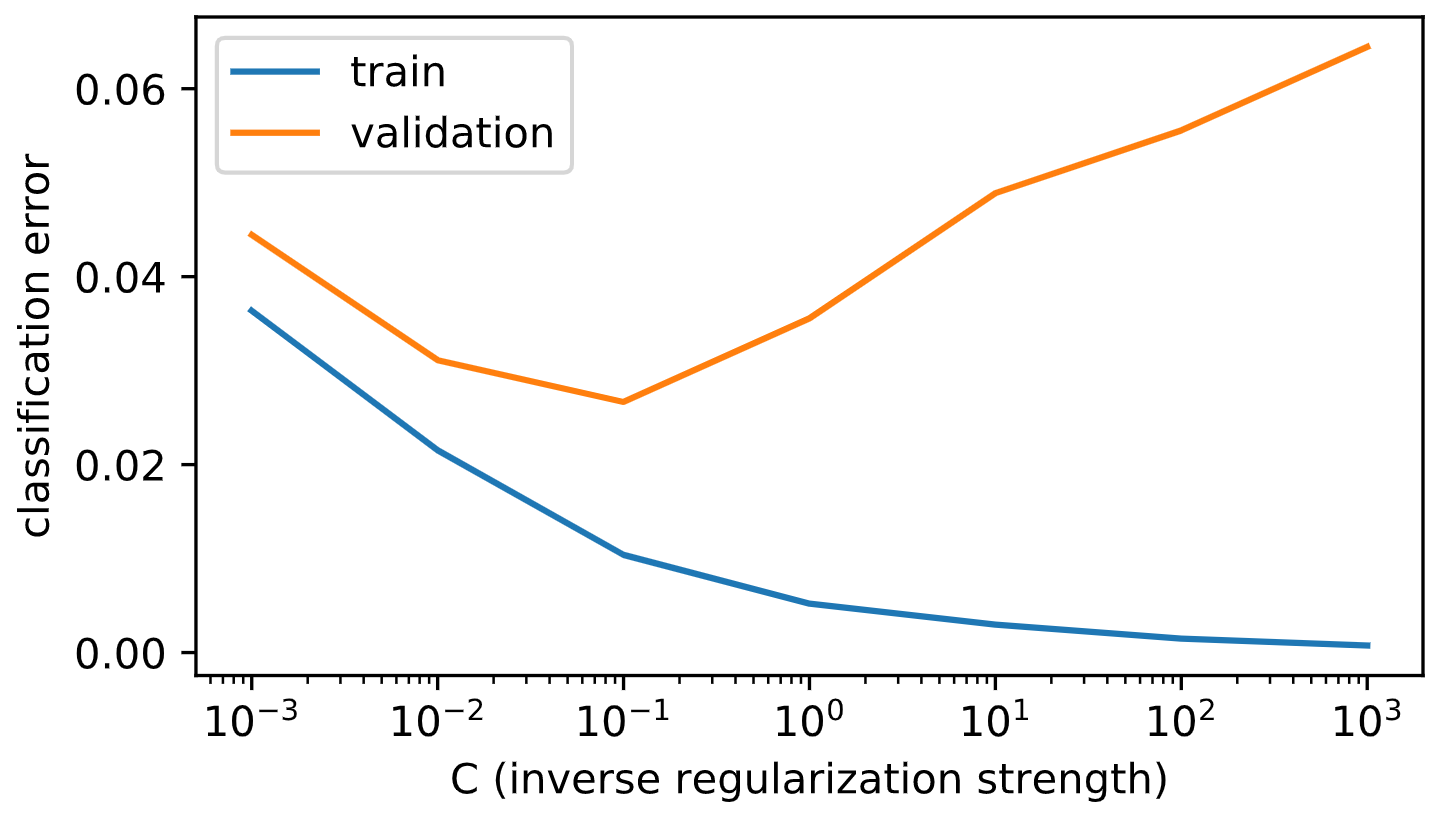
plt.legend(("train", "validation"))

plt.show()

**Incorrect Submission**

Did you correctly specify the body? Check your call of LogisticRegression(). Did you specify the argument C?

Check the first for loop. Did you correctly specify the body? Have you specified the arguments for lr.score() using the right syntax?



Traceback (most recent call last):

File "script.py", line 12, in <module>

train\_errs.append( 1.0 - lr.score(X\_valid) )

TypeError: score() missing 1 required positional argument: 'y'

Traceback (most recent call last):

File "script.py", line 9, in <module>

lr.fit(X\_train, y\_train)

File "script.py", line 1233, in fit

sample\_weight=sample\_weight)

File "script.py", line 886, in \_fit\_liblinear

solver\_type = \_get\_liblinear\_solver\_type(multi\_class, penalty, loss, dual)

File "script.py", line 747, in \_get\_liblinear\_solver\_type

% (error\_string, penalty, loss, dual))

ValueError: Unsupported set of arguments: The combination of penalty='0.001' and loss='logistic\_regression' is not supported, Parameters: penalty=0.001, loss='logistic\_regression', dual=False

Traceback (most recent call last):

File "script.py", line 8, in <module>

lr = LogisticRegression(C)

NameError: name 'C' is not defined

Traceback (most recent call last):

File "script.py", line 12, in <module>

train\_errs.append( 1.0 - lr.score(X\_valid) )

TypeError: score() missing 1 required positional argument: 'y'

In [1]:

+70 XP

Congrats! As you can see, too much regularization (small C) doesn't work well - due to underfitting - and too little regularization (large C) doesn't work well either - due to overfitting.

**Exercise**

**Exercise**

**Logistic regression and feature selection**

In this exercise we'll perform feature selection on the movie review sentiment data set using L1 regularization. The features and targets are already loaded for you in X\_train and y\_train.

We'll search for the best value of C using scikit-learn's GridSearchCV(), which was covered in the prerequisite course.

**Instructions**

**100 XP**

* Instantiate a logistic regression object that uses L1 regularization.
* Find the value of C that minimizes cross-validation error.
* Print out the number of selected features for this value of C.

[**Take Hint (-30 XP)**](javascript:void(0))

[**Show Answer (-70 XP)**](javascript:void(0))

**Hint**

* You can set the type of regularization in a LogisticRegression using the penalty parameter. Here, it should be a lowercase l1.
* best\_lr is a regular logistic regression object. How do we usually get the coefficients of a model?

# Specify L1 regularization

lr = LogisticRegression(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))

In [1]: best\_lr

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

best\_lr

NameError: name 'best\_lr' is not defined

In [2]: best\_lr

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

best\_lr

NameError: name 'best\_lr' is not defined

In [3]: # Specify L1 regularization

lr = LogisticRegression(\_\_\_\_)

# 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.\_\_\_\_

print("Total number of features:", coefs.size)

print("Number of selected features:", np.count\_nonzero(coefs))

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

lr = LogisticRegression(\_\_\_\_)

NameError: name '\_\_\_\_' is not defined

In [4]: # Specify L1 regularization

lr = LogisticRegression(C=0.001)

# 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.\_\_\_\_

print("Total number of features:", coefs.size)

print("Number of selected features:", np.count\_nonzero(coefs))

Best CV params {'C': 0.1}

Traceback (most recent call last):

File "<stdin>", line 13, in <module>

coefs = best\_lr.\_\_\_\_

AttributeError: 'LogisticRegression' object has no attribute '\_\_\_\_'

In [5]: dir(best\_lr)

Out[5]:

['C',

'\_\_class\_\_',

'\_\_delattr\_\_',

'\_\_dict\_\_',

'\_\_dir\_\_',

'\_\_doc\_\_',

'\_\_eq\_\_',

'\_\_format\_\_',

'\_\_ge\_\_',

'\_\_getattribute\_\_',

'\_\_getstate\_\_',

'\_\_gt\_\_',

'\_\_hash\_\_',

'\_\_init\_\_',

'\_\_init\_subclass\_\_',

'\_\_le\_\_',

'\_\_lt\_\_',

'\_\_module\_\_',

'\_\_ne\_\_',

'\_\_new\_\_',

'\_\_reduce\_\_',

'\_\_reduce\_ex\_\_',

'\_\_repr\_\_',

'\_\_setattr\_\_',

'\_\_setstate\_\_',

'\_\_sizeof\_\_',

'\_\_str\_\_',

'\_\_subclasshook\_\_',

'\_\_weakref\_\_',

'\_estimator\_type',

'\_get\_param\_names',

'\_predict\_proba\_lr',

'class\_weight',

'classes\_',

'coef\_',

'decision\_function',

'densify',

'dual',

'fit',

'fit\_intercept',

'get\_params',

'intercept\_',

'intercept\_scaling',

'max\_iter',

'multi\_class',

'n\_iter\_',

'n\_jobs',

'penalty',

'predict',

'predict\_log\_proba',

'predict\_proba',

'random\_state',

'score',

'set\_params',

'solver',

'sparsify',

'tol',

'verbose',

'warm\_start']

In [6]:

<script.py> output:

Best CV params {'C': 0.1}

Total number of features: 2500

Number of selected features: 2500

Traceback (most recent call last):

File "script.py", line 2, in <module>

lr = LogisticRegression(penalty=l1)

NameError: name 'l1' is not defined

<script.py> output:

Best CV params {'C': 1}

Total number of features: 2500

Number of selected features: 1220

In [6]:

**Incorrect Submission**

Check your call of LogisticRegression(). Did you specify the argument penalty?

Did you define the variable lr without errors?

+70 XP

Great job! As you can see, a whole lot of features were discarded here.

**Exercise**

**Exercise**

**Identifying the most positive and negative words**

In this exercise we'll try to interpret the coefficients of a logistic regression fit on the movie review sentiment dataset. The model object is already instantiated and fit for you in the variable lr.

In addition, the words corresponding to the different features are loaded into the variable vocab. For example, since vocab[100] is "think", that means feature 100 corresponds to the number of times the word "think" appeared in that movie review.

**Instructions**

**100 XP**

* Find the words corresponding to the 5 largest coefficients.
* Find the words corresponding to the 5 smallest coefficients.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Check your call of print(). Did you correctly specify the first argument? Expected loved, but got 813.

Check your call of print(). Did you correctly specify the first argument? Expected loved, but got to.

Check your call of print(). Did you correctly specify the first argument? Expected loved, but got lame.

# Get the indices of the sorted cofficients

inds\_ascending = np.argsort(lr.coef\_.flatten())

inds\_descending = inds\_ascending[::-1]

# Print the most positive words

print("Most positive words: ", end="")

for i in range(5):

print(vocab[inds\_descending[i]], end=", ")

print("\n")

# Print most negative words

print("Most negative words: ", end="")

for i in range(5):

print(vocab[inds\_ascending[i]], end=", ")

print("\n")

<script.py> output:

Most positive words: 1278, 427, 240, 344, 813,

Most negative words: 493, 870, 1458, 1228, 437,

<script.py> output:

Most positive words: the, and, a, of, to,

Most negative words: the, and, a, of, to,

<script.py> output:

Most positive words: disappointing, waste, worst, boring, lame,

Most negative words: favorite, superb, noir, knowing, loved,

In [1]: vocab.find('loved')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

vocab.find('loved')

AttributeError: 'numpy.ndarray' object has no attribute 'find'

In [2]: vocab.tolist().find('loved')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

vocab.tolist().find('loved')

AttributeError: 'list' object has no attribute 'find'

In [3]: vocab.tolist().index('loved')

Out[3]: 437

In [4]: vocab

Out[4]: array(['the', 'and', 'a', ..., 'birth', 'sorts', 'gritty'], dtype='<U14')

In [5]: vocab[1278]

Out[5]: 'disappointing'

In [6]: vocab[493]

Out[6]: 'favorite'

<script.py> output:

Most positive words: favorite, superb, noir, knowing, loved,

Most negative words: disappointing, waste, worst, boring, lame,

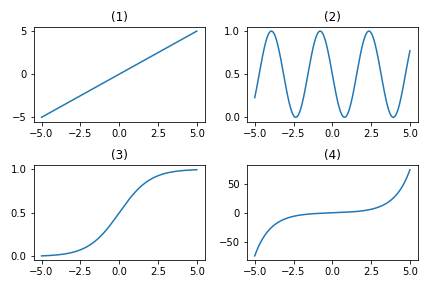
In [7]:

+100 XP

You got it! The answers sort of make sense, don't they?

**Getting class probabilities**

Which of the following transformations would make sense for transforming the raw model output of a linear classifier into a class probability?



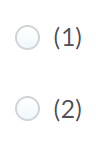
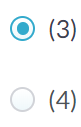
**Answer the question**

**50 XP**

**Possible Answers**

**Incorrect Submission**

Look at the y-axis labels. Remember that probabilities are between 0 and 1.

+50 XP

That's right! The function in the picture is fairly similar to the logistic function used by logistic regression.

**Exercise**

**Exercise**

**Regularization and probabilities**

In this exercise, you will observe the effects of changing the regularization strength on the predicted probabilities.

A 2D binary classification dataset is already loaded into the environment as X and y.

**Instructions 1/2**

**50 XP**

* [1](javascript:void(0))
  + Compute the maximum predicted probability.
  + Run the provided code and take a look at the plot.

[**Take Hint (-15 XP)**](javascript:void(0))

* [2](javascript:void(0))
  + Create a model with C=0.1 and examine how the plot and probabilities change.

# Set the regularization strength

model = LogisticRegression(C=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))

# Set the regularization strength

model = LogisticRegression(C=0.1)

# Fit and plot

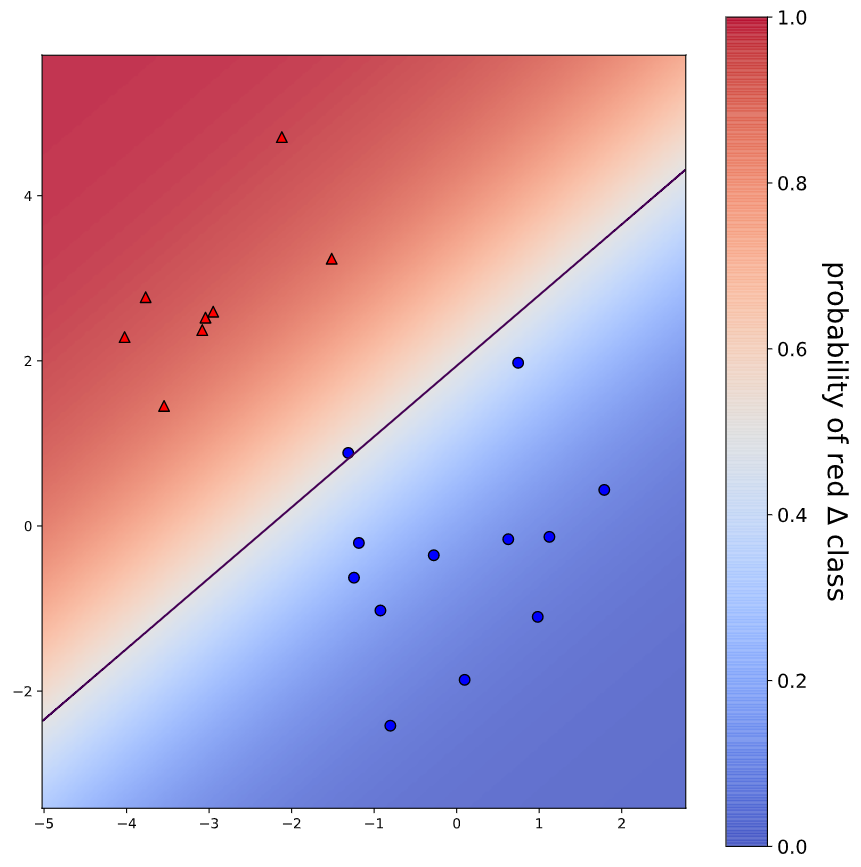
model.fit(X,y)

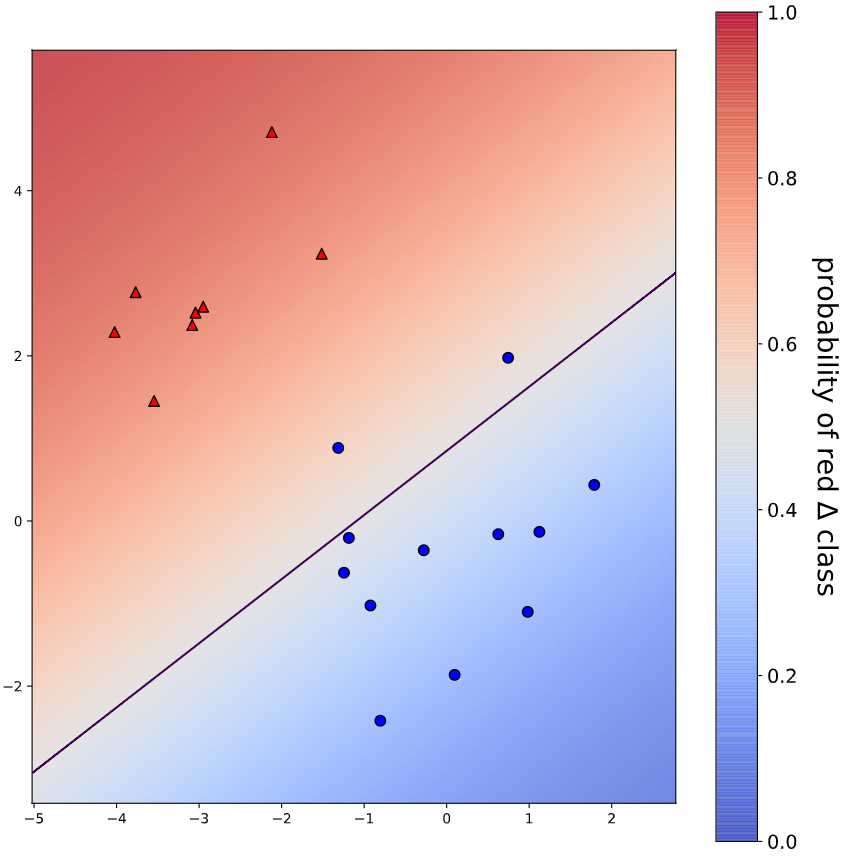
plot\_classifier(X,y,model,proba=True)

# Predict probabilities on training points

prob = model.predict\_proba(X)

print("Maximum predicted probability", np.max(prob))





In [1]: prob)

File "<stdin>", line 1

prob)

^

SyntaxError: invalid syntax

In [2]: # Set the regularization strength

model = LogisticRegression(C=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", max(prob))

Traceback (most recent call last):

File "<stdin>", line 10, in <module>

print("Maximum predicted probability", max(prob))

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()

In [3]: # Set the regularization strength

model = LogisticRegression(C=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))

Maximum predicted probability 0.9761229966765974

<script.py> output:

Maximum predicted probability 0.9761229966765974

<script.py> output:

Maximum predicted probability 0.8990965659596716

In [4]:

+100 XP

You got it! As you probably noticed, smaller values of C lead to less confident predictions. That's because smaller C means more regularization, which in turn means smaller coefficients, which means raw model outputs closer to zero and, thus, probabilities closer to 0.5 after the raw model output is squashed through the sigmoid function. That's quite a chain of events!

**Exercise**

**Exercise**

**Visualizing easy and difficult examples**

In this exercise, you'll visualize the examples that the logistic regression model is most and least confident about by looking at the largest and smallest predicted probabilities.

The handwritten digits dataset is already loaded into the variables X and y. The show\_digit function takes in an integer index and plots the corresponding image, with some extra information displayed above the image.

**Instructions**

**100 XP**

* Fill in the first blank with the *index* of the digit that the model is most confident about.
* Fill in the second blank with the *index* of the digit that the model is least confident about.
* Observe the images: do you agree that the first one is less ambiguous than the second?

[**Take Hint (-30 XP)**](javascript:void(0))

[**Show Answer (-70 XP)**](javascript:void(0))

**Hint**

proba\_inds contains the information you need. The first element of proba\_inds corresponds to the index of the digit that the model is most least confident about. The second element corresponds to the index of the digit that the model is next least confident about. Etc.

**Incorrect Submission**

Check your call of show\_digit(). Did you correctly specify the first argument? Expected proba\_inds[-1], but got proba\_inds[0].

lr = LogisticRegression()

lr.fit(X,y)

# Get predicted probabilities

proba = lr.predict\_proba(X)

# Sort the example indices by their maximum probability

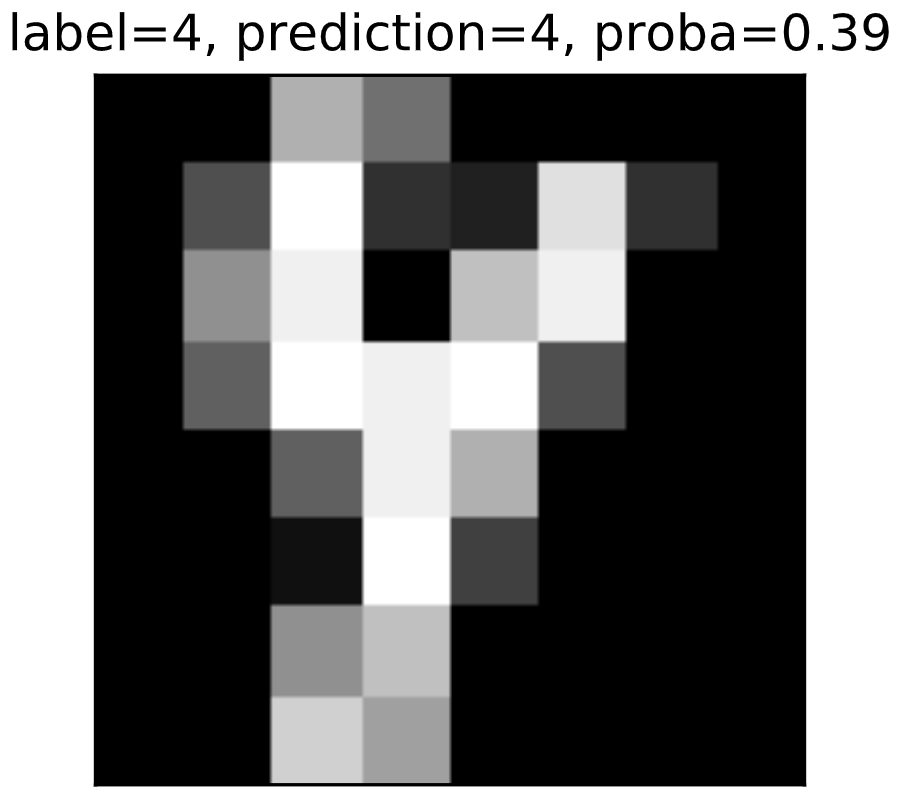
proba\_inds = np.argsort(np.max(proba,axis=1))

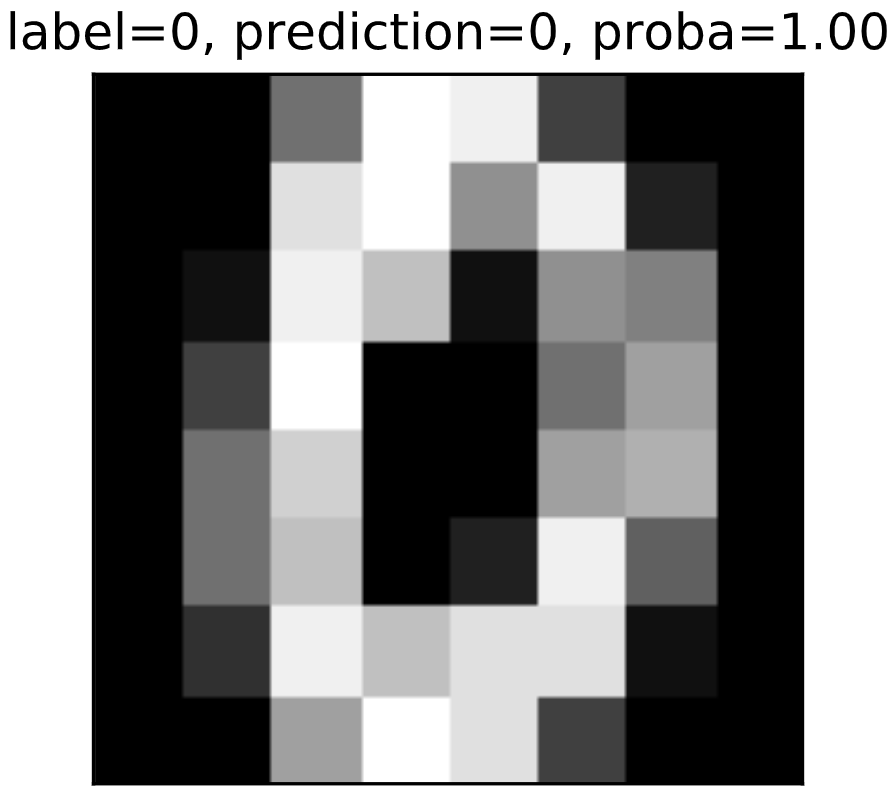
# Show the most confident (least ambiguous) digit

show\_digit(proba\_inds[-1], lr)

# Show the least confident (most ambiguous) digit

show\_digit(proba\_inds[0], lr)





In [1]: show\_digit

Out[1]: <function \_\_main\_\_.show\_digit>

In [2]: help(np.argsort)

Help on function argsort in module numpy:

argsort(a, axis=-1, kind=None, order=None)

Returns the indices that would sort an array.

Perform an indirect sort along the given axis using the algorithm specified

by the `kind` keyword. It returns an array of indices of the same shape as

`a` that index data along the given axis in sorted order.

Parameters

----------

a : array\_like

Array to sort.

axis : int or None, optional

Axis along which to sort. The default is -1 (the last axis). If None,

the flattened array is used.

kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, optional

Sorting algorithm. The default is 'quicksort'. Note that both 'stable'

and 'mergesort' use timsort under the covers and, in general, the

actual implementation will vary with data type. The 'mergesort' option

is retained for backwards compatibility.

.. versionchanged:: 1.15.0.

The 'stable' option was added.

order : str or list of str, optional

When `a` is an array with fields defined, this argument specifies

which fields to compare first, second, etc. A single field can

be specified as a string, and not all fields need be specified,

but unspecified fields will still be used, in the order in which

they come up in the dtype, to break ties.

Returns

-------

index\_array : ndarray, int

Array of indices that sort `a` along the specified `axis`.

If `a` is one-dimensional, ``a[index\_array]`` yields a sorted `a`.

More generally, ``np.take\_along\_axis(a, index\_array, axis=axis)``

always yields the sorted `a`, irrespective of dimensionality.

See Also

--------

sort : Describes sorting algorithms used.

lexsort : Indirect stable sort with multiple keys.

ndarray.sort : Inplace sort.

argpartition : Indirect partial sort.

Notes

-----

See `sort` for notes on the different sorting algorithms.

As of NumPy 1.4.0 `argsort` works with real/complex arrays containing

nan values. The enhanced sort order is documented in `sort`.

Examples

--------

One dimensional array:

>>> x = np.array([3, 1, 2])

>>> np.argsort(x)

array([1, 2, 0])

Two-dimensional array:

>>> x = np.array([[0, 3], [2, 2]])

>>> x

array([[0, 3],

[2, 2]])

>>> ind = np.argsort(x, axis=0) # sorts along first axis (down)

>>> ind

array([[0, 1],

[1, 0]])

>>> np.take\_along\_axis(x, ind, axis=0) # same as np.sort(x, axis=0)

array([[0, 2],

[2, 3]])

>>> ind = np.argsort(x, axis=1) # sorts along last axis (across)

>>> ind

array([[0, 1],

[0, 1]])

>>> np.take\_along\_axis(x, ind, axis=1) # same as np.sort(x, axis=1)

array([[0, 3],

[2, 2]])

Indices of the sorted elements of a N-dimensional array:

>>> ind = np.unravel\_index(np.argsort(x, axis=None), x.shape)

>>> ind

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

>>> x[ind] # same as np.sort(x, axis=None)

array([0, 2, 2, 3])

Sorting with keys:

>>> x = np.array([(1, 0), (0, 1)], dtype=[('x', '<i4'), ('y', '<i4')])

>>> x

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

dtype=[('x', '<i4'), ('y', '<i4')])

>>> np.argsort(x, order=('x','y'))

array([1, 0])

>>> np.argsort(x, order=('y','x'))

array([0, 1])

In [3]:

+70 XP

Great job! As you can see, the least confident example looks like a weird 4, and the most confident example looks like a very typical 0.

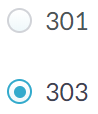
**Counting the coefficients**

If you fit a logistic regression model on a classification problem with 3 classes and 100 features, how many coefficients would you have, including intercepts?

**Answer the question**

**50 XP**

**Possible Answers**

**Incorrect Submission**

Close! But each class gets its own intercept.

Reply: I was hesitated you …!

Awesome, thanks for your feedback!

+50 XP

Nicely done! Feel free to test this out with scikit-learn!

**Exercise**

**Exercise**

**Fitting multi-class logistic regression**

In this exercise, you'll fit the two types of multi-class logistic regression, one-vs-rest and softmax/multinomial, on the handwritten digits data set and compare the results. The handwritten digits dataset is already loaded and split into X\_train, y\_train, X\_test, and y\_test.

**Instructions**

**100 XP**

* Fit a one-vs-rest logistic regression classifier and report the results.
* Fit a multinomial logistic regression classifier by setting the multi\_class parameter, plus setting to be solver = "lbfgs", and report the results.

[**Take Hint (-30 XP)**](javascript:void(0))

[**Show Answer (-70 XP)**](javascript:void(0))

**Hint**

* LogisticRegression uses a one-vs-rest classifier by default, so there's no need to change any settings for the first model.
* To instantiate a softmax/multinomial classifier, you need to set the multi\_class and solver parameters. The multi\_class parameter needs to be set to "multinomial".

 Awesome, thanks for your feedback!

**Incorrect Submission**

Check your second call of LogisticRegression(). Did you correctly specify the argument solver? Expected "lbfgs", but got 'softmax'.

# Fit one-vs-rest logistic regression classifier

lr\_ovr = LogisticRegression()

lr\_ovr.fit(X\_train, y\_train)

print("OVR training accuracy:", lr\_ovr.score(X\_train, y\_train))

print("OVR test accuracy :", lr\_ovr.score(X\_test, y\_test))

# Fit softmax classifier

lr\_mn = LogisticRegression(multi\_class="multinomial", solver='lbfgs')

lr\_mn.fit(X\_train, y\_train)

print("Softmax training accuracy:", lr\_mn.score(X\_train, y\_train))

print("Softmax test accuracy :", lr\_mn.score(X\_test, y\_test))

In [1]: softmax

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

softmax

NameError: name 'softmax' is not defined

In [2]: LogisticRegression

Out[2]: sklearn.linear\_model.logistic.LogisticRegression

<script.py> output:

OVR training accuracy: 0.9948032665181886

OVR test accuracy : 0.9644444444444444

Traceback (most recent call last):

File "script.py", line 10, in <module>

lr\_mn.fit(X\_train, y\_train)

File "script.py", line 1222, in fit

self.dual)

File "script.py", line 430, in \_check\_solver\_option

% solver)

ValueError: Logistic Regression supports only liblinear, newton-cg, lbfgs, sag and saga solvers, got softmax

<script.py> output:

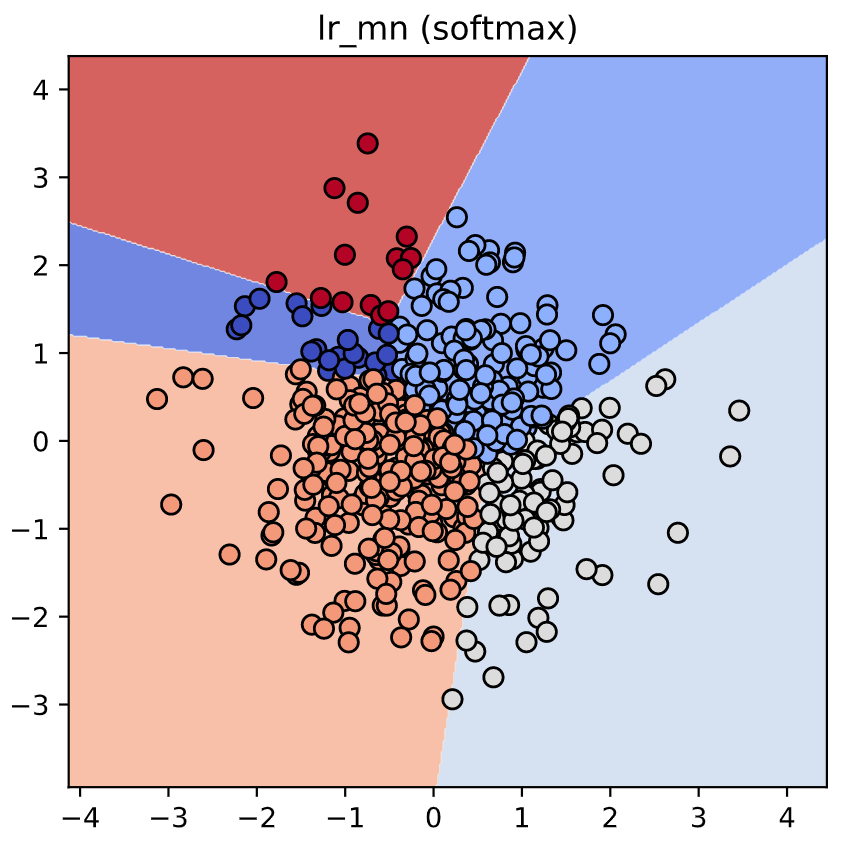
OVR training accuracy: 0.9948032665181886

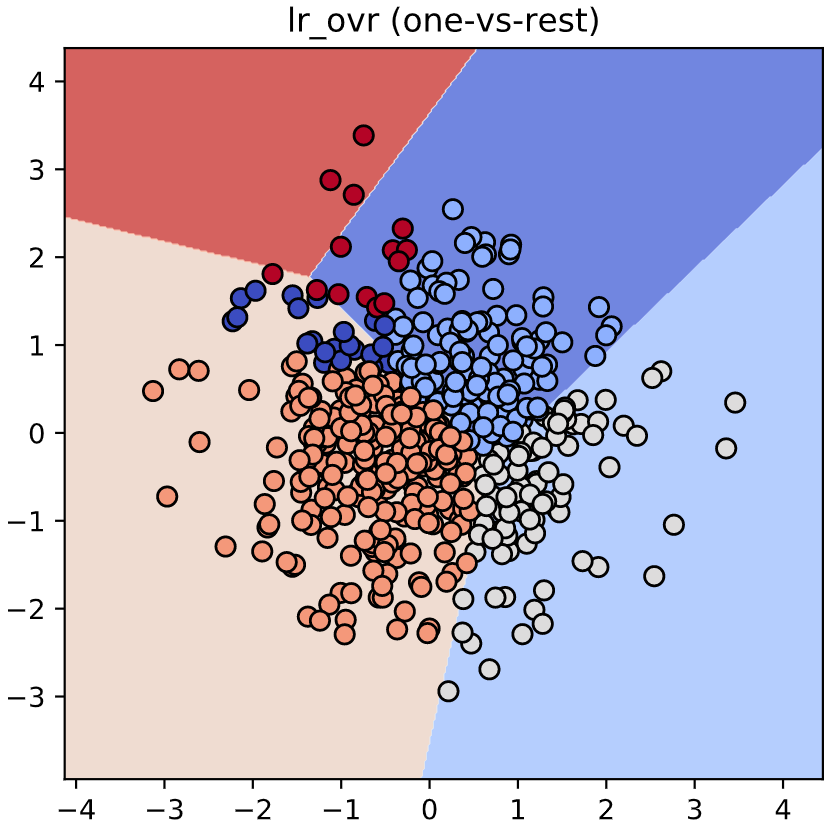
OVR test accuracy : 0.9644444444444444

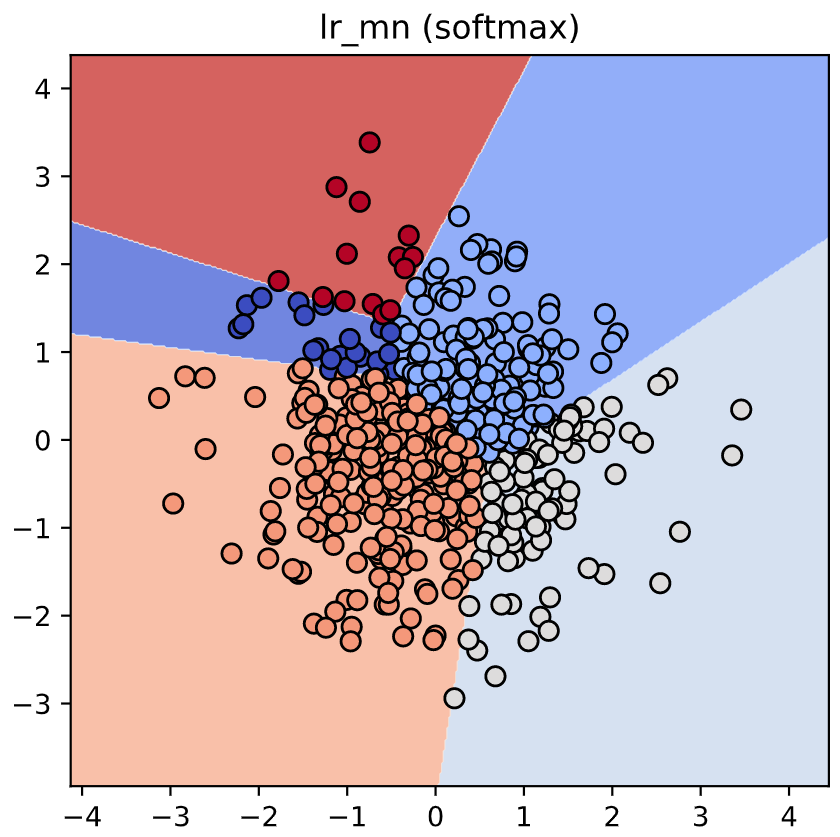
Softmax training accuracy: 1.0

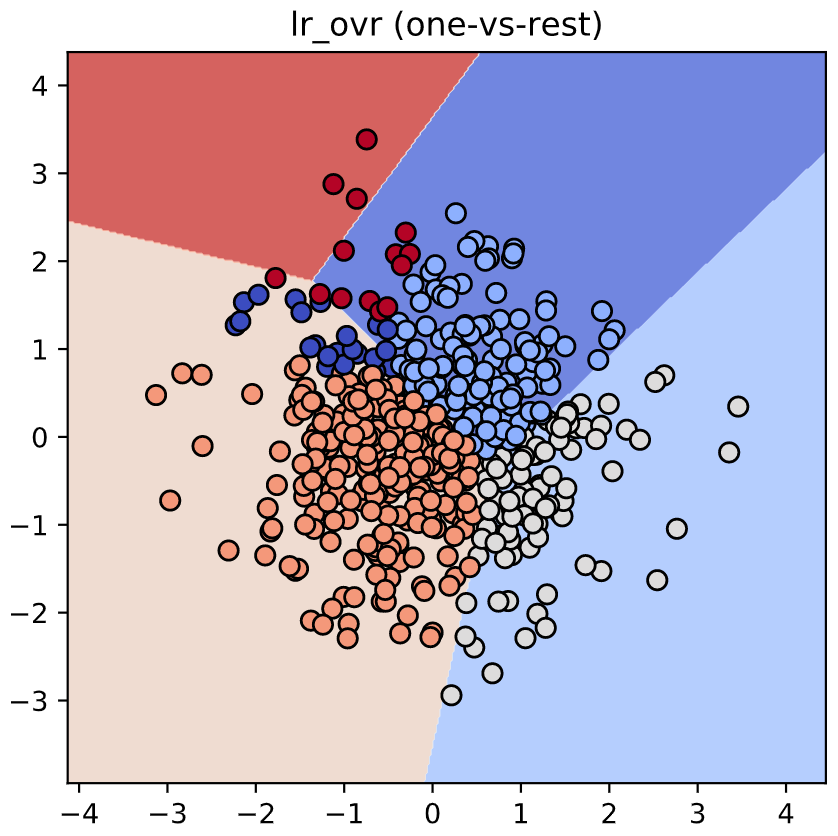
Softmax test accuracy : 0.9688888888888889

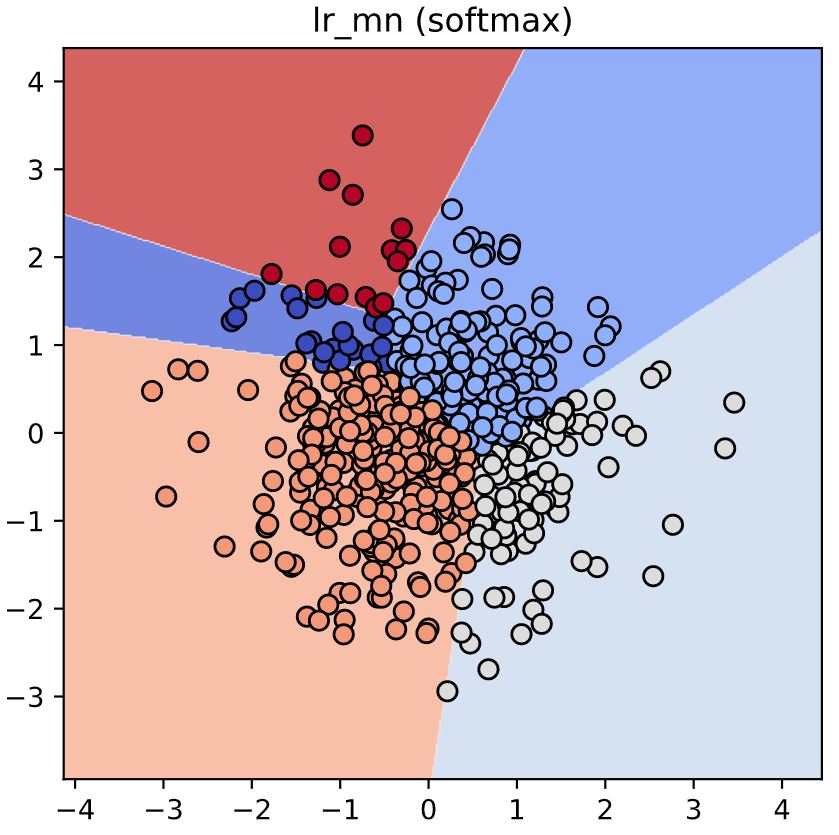
In [3]:











+70 XP

Nice work! As you can see, the accuracies of the two methods are fairly similar on this data set.

**Exercise**

**Exercise**

**Visualizing multi-class logistic regression**

In this exercise we'll continue with the two types of multi-class logistic regression, but on a toy 2D data set specifically designed to break the one-vs-rest scheme.

The data set is loaded into X\_train and y\_train. The two logistic regression objects,lr\_mn and lr\_ovr, are already instantiated (with C=100), fit, and plotted.

Notice that lr\_ovr never predicts the dark blue class... yikes! Let's explore why this happens by plotting one of the binary classifiers that it's using behind the scenes.

**Instructions**

**100 XP**

* Create a new logistic regression object (also with C=100) to be used for binary classification.
* Visualize this binary classifier with plot\_classifier... does it look reasonable?

[**Take Hint (-30 XP)**](javascript:void(0))

[**Show Answer (-70 XP)**](javascript:void(0))

**Hint**

* Create a logistic regression object with C=100 and the default settings for other hyperparameters.
* Then, pass this object, named lr\_class\_1, into plot\_classifier.

 Awesome, thanks for your feedback!

**Incorrect Submission**

Check your call of LogisticRegression(). Did you specify the argument C?

# Print training accuracies

print("Softmax training accuracy:", lr\_mn.score(X\_train, y\_train))

print("One-vs-rest training accuracy:", lr\_ovr.score(X\_train, y\_train))

# Create the binary classifier (class 1 vs. rest)

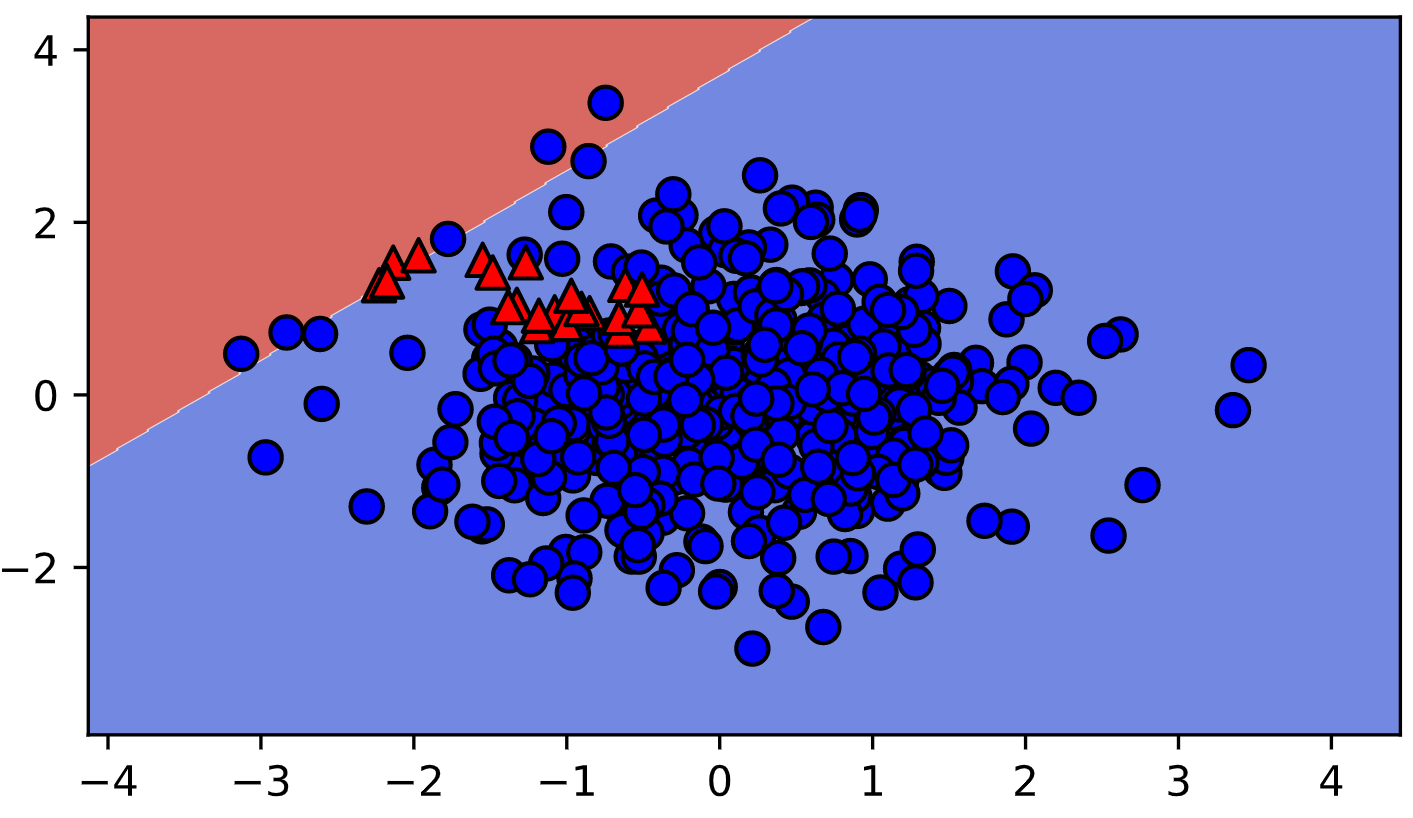
lr\_class\_1 = LogisticRegression(C=100)

lr\_class\_1.fit(X\_train, y\_train==1)

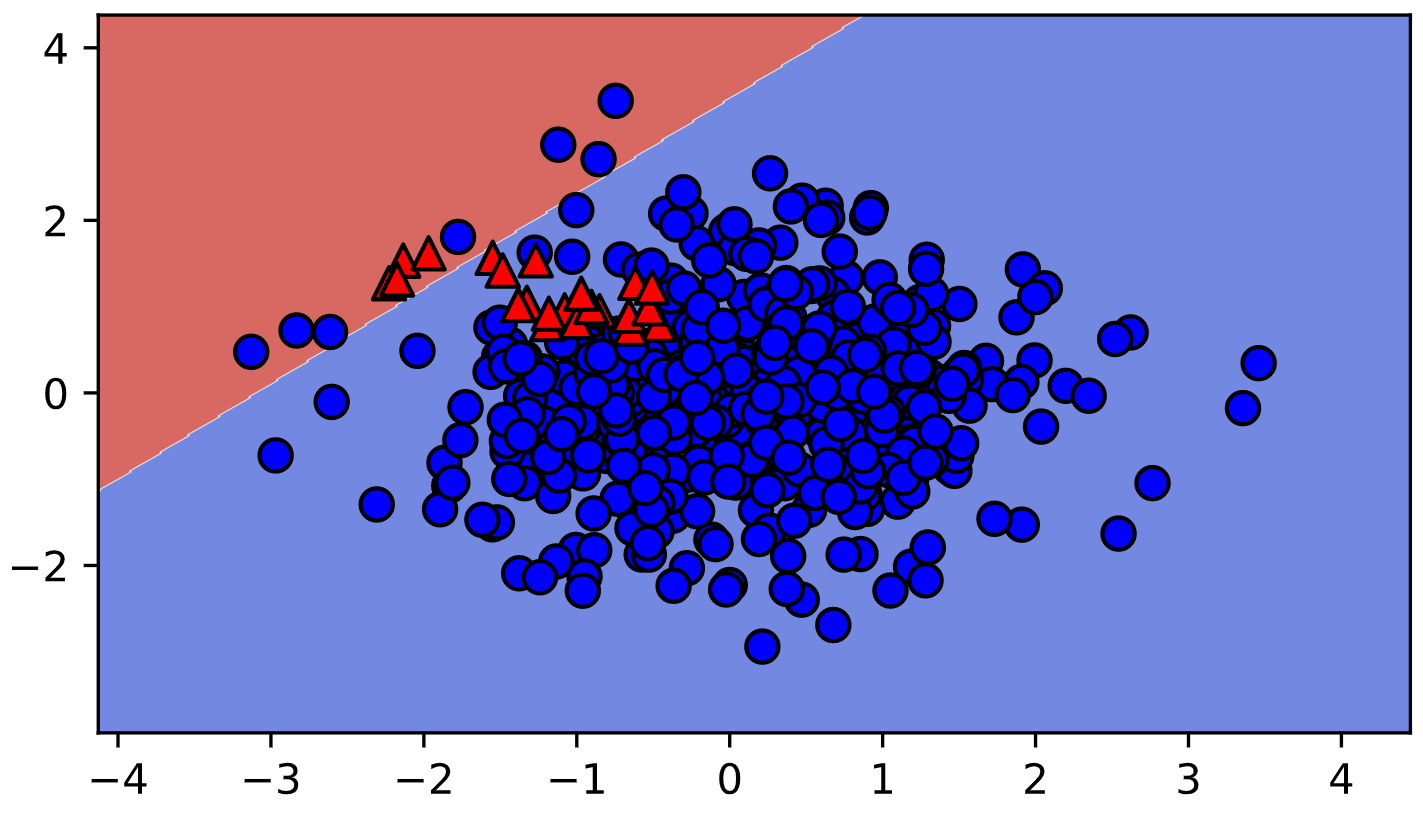
# Plot the binary classifier (class 1 vs. rest)

plot\_classifier(X\_train, y\_train==1, svm\_class\_1)

C=1:



C=10:



<script.py> output:

Softmax training accuracy: 0.996

One-vs-rest training accuracy: 0.916

<script.py> output:

Softmax training accuracy: 0.996

One-vs-rest training accuracy: 0.916

In [1]:

+70 XP

Nice work! As you can see, the binary classifier incorrectly labels almost all points in class 1 (shown as red triangles in the final plot)! Thus, this classifier is not a very effective component of the one-vs-rest classifier. In general, though, one-vs-rest often works well.

**Exercise**

**Exercise**

**One-vs-rest SVM**

As motivation for the next and final chapter on support vector machines, we'll repeat the previous exercise with a non-linear SVM. Once again, the data is loaded into X\_train, y\_train, X\_test, and y\_test .

Instead of using LinearSVC, we'll now use scikit-learn's SVC object, which is a non-linear "kernel" SVM (much more on what this means in Chapter 4!). Again, your task is to create a plot of the binary classifier for class 1 vs. rest.

**Instructions**

**100 XP**

* Fit an SVC called svm\_class\_1 to predict class 1 vs. other classes.
* Plot this classifier.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Check your call of svm\_class\_1.fit(). Did you correctly specify the second argument? Expected y\_train==1, but got y\_train.

Awesome, thanks for your feedback!

# We'll use SVC instead of LinearSVC from now on

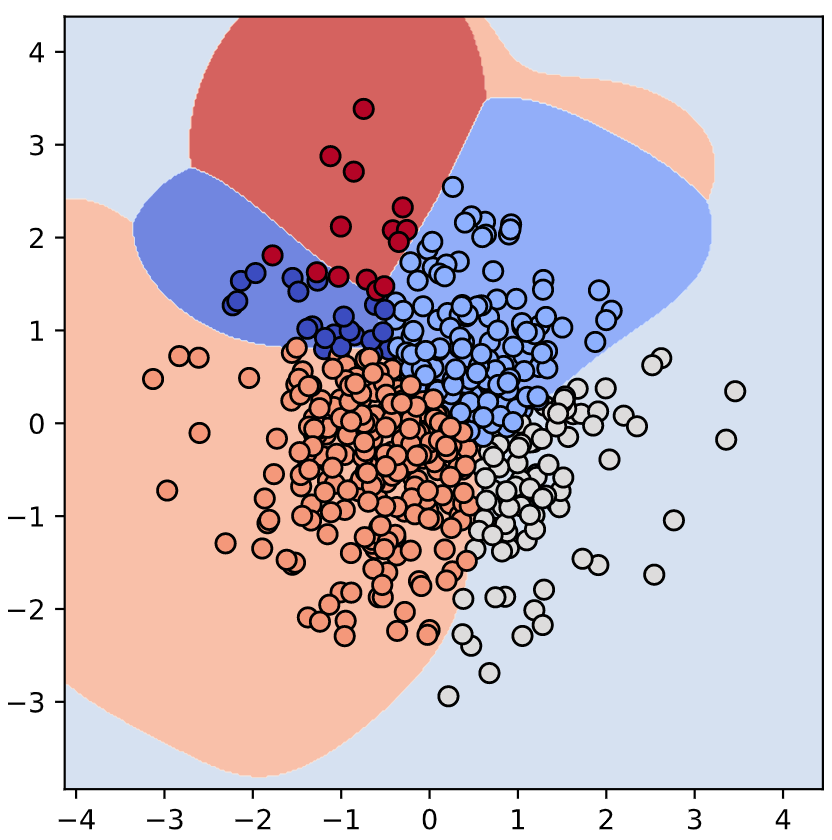
from sklearn.svm import SVC

# Create/plot the binary classifier (class 1 vs. rest)

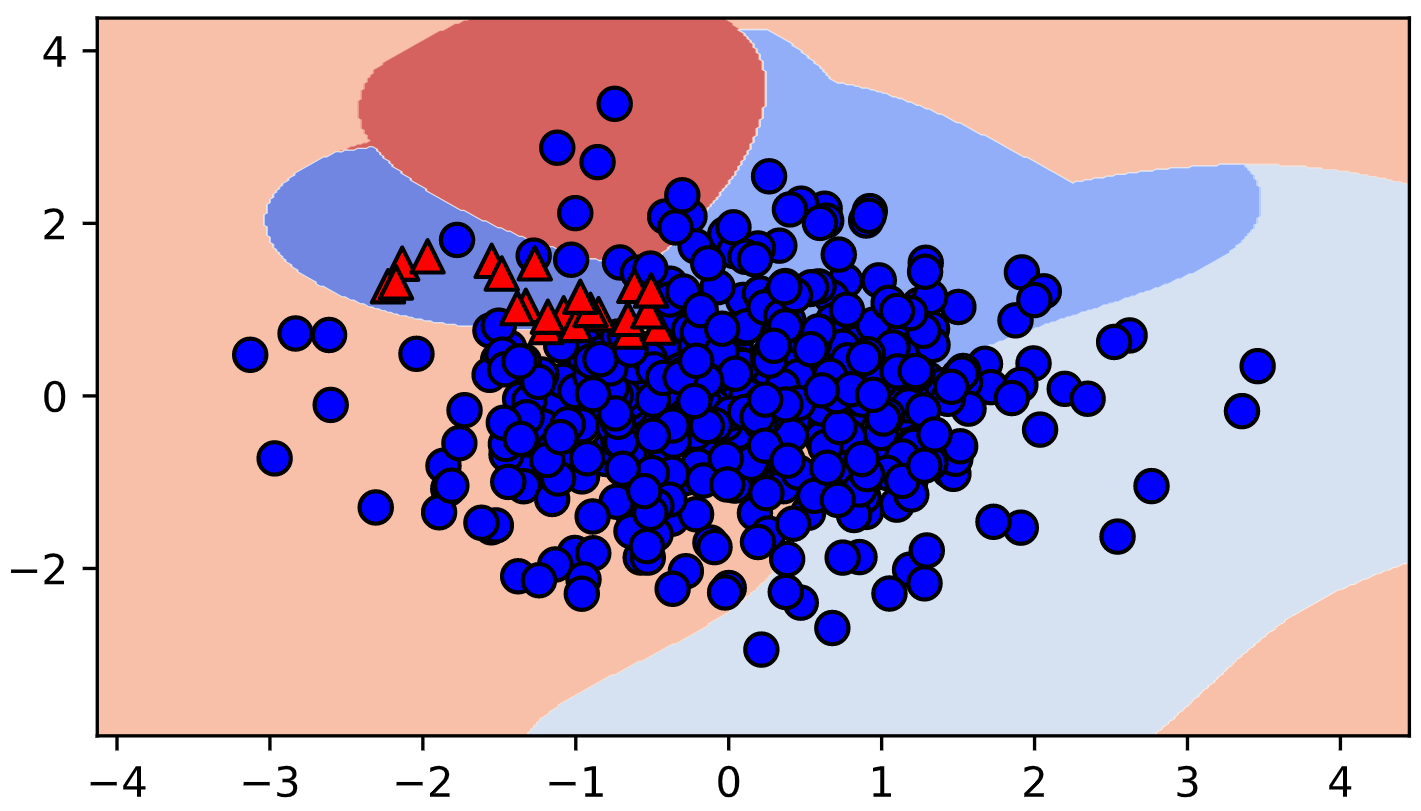
svm\_class\_1 = SVC()

svm\_class\_1.fit(X\_train, y\_train==1)

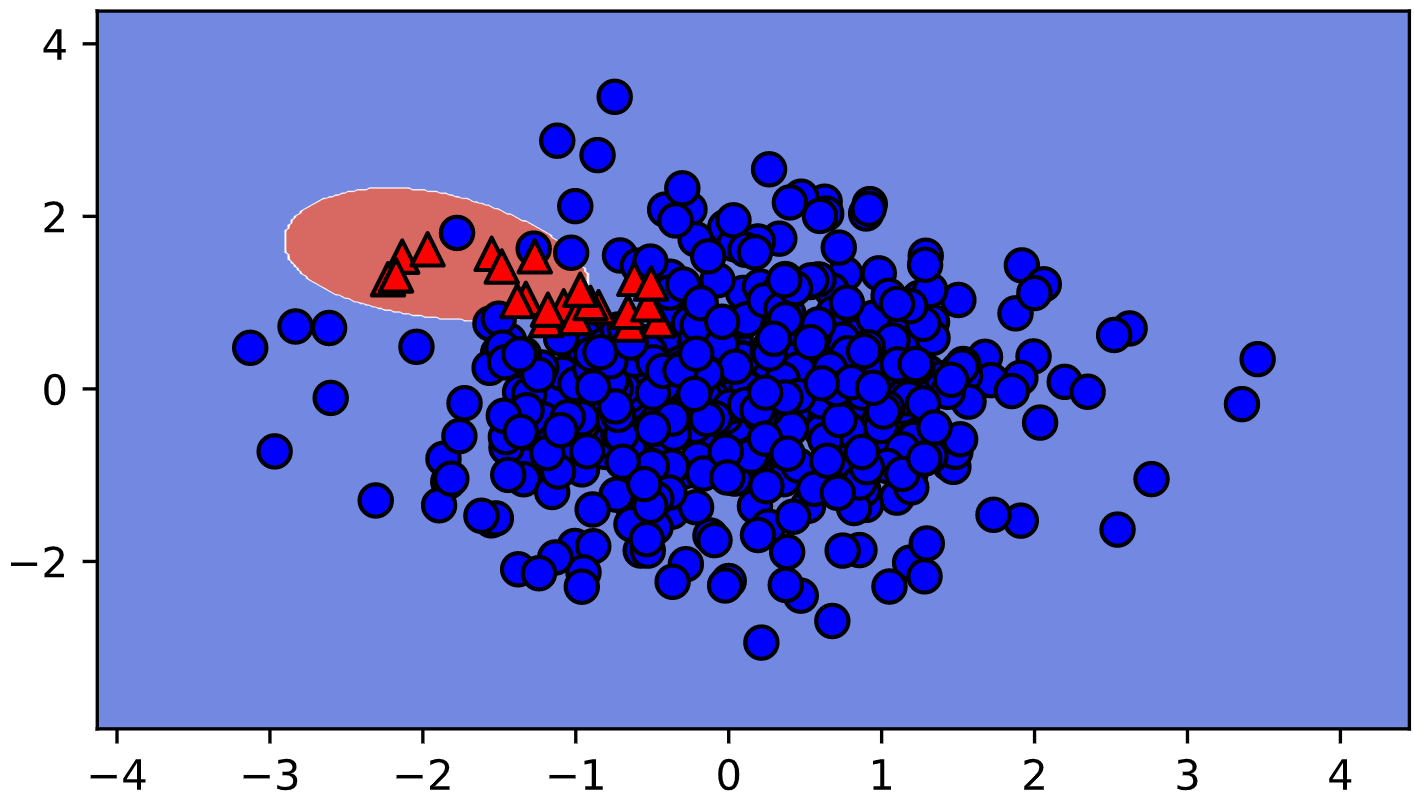
plot\_classifier(X\_train, y\_train==1, svm\_class\_1)



y\_train only:



y\_train == 1:



Traceback (most recent call last):

File "script.py", line 7, in <module>

plot\_classifier(X\_train, y\_train==1, lr\_class\_1)

NameError: name 'lr\_class\_1' is not defined

In [1]:

+100 XP

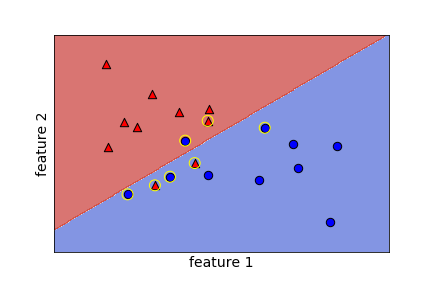
Cool, eh?! The non-linear SVM works fine with one-vs-rest on this dataset because it learns to "surround" class 1.

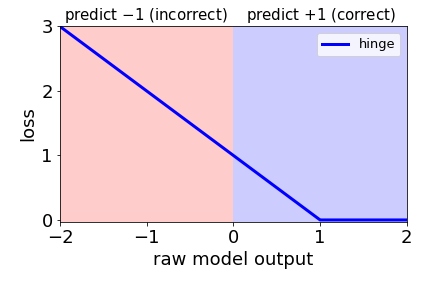
+50 XP

Please give a rating: 5

**Support vector definition**

Which of the following is a true statement about support vectors? To help you out, here's the picture of support vectors from the video (top), as well as the hinge loss from Chapter 2 (bottom).





**Answer the question**

**35 XP**

**Possible Answers**

All support vectors are classified correctly.

All support vectors are classified incorrectly.

All correctly classified points are support vectors.

All incorrectly classified points are support vectors.

**Hint**

There are two ways to become a support vector: either the point is classified incorrectly, or it is classified correctly but very close to the boundary.

**Incorrect Submission**

Are you sure about that? Take another look at the figure above.

Support vectors are the points that "matter". They aren't necessarily classified correctly.

Almost! But one can also become a support vector by being classified correctly and close to the boundary.

+35 XP

Nice work, you got it!

**Exercise**

**Exercise**

**Effect of removing examples**

Support vectors are defined as training examples that influence the decision boundary. In this exercise, you'll observe this behavior by removing non support vectors from the training set.

The wine quality dataset is already loaded into X and y (first two features only). (Note: we specify lims in plot\_classifier() so that the two plots are forced to use the same axis limits and can be compared directly.)

**Instructions**

**100 XP**

* Train a linear SVM on the whole data set.
* Create a new data set containing only the support vectors.
* Train a new linear SVM on the smaller data set.

[**Take Hint (-30 XP)**](javascript:void(0))

[**Show Answer (-70 XP)**](javascript:void(0))

**Hint**

svm.support\_ returns the indices of the support vectors of the model svm. You can use this to directly index into X and y and return subsets containly only those examples. For example, X[svm.support\_].

# 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\_]

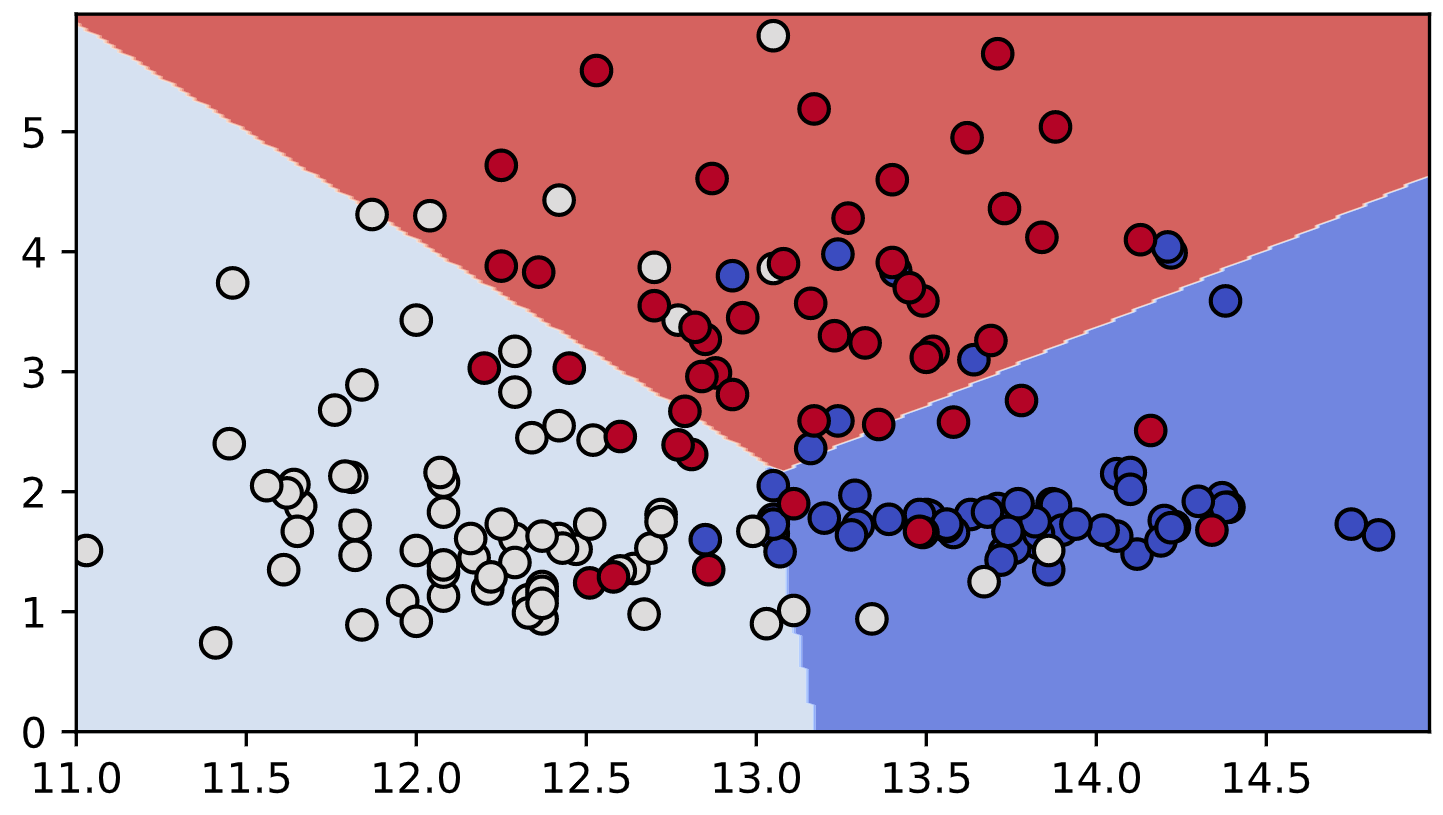
y\_small = y[svm.support\_]

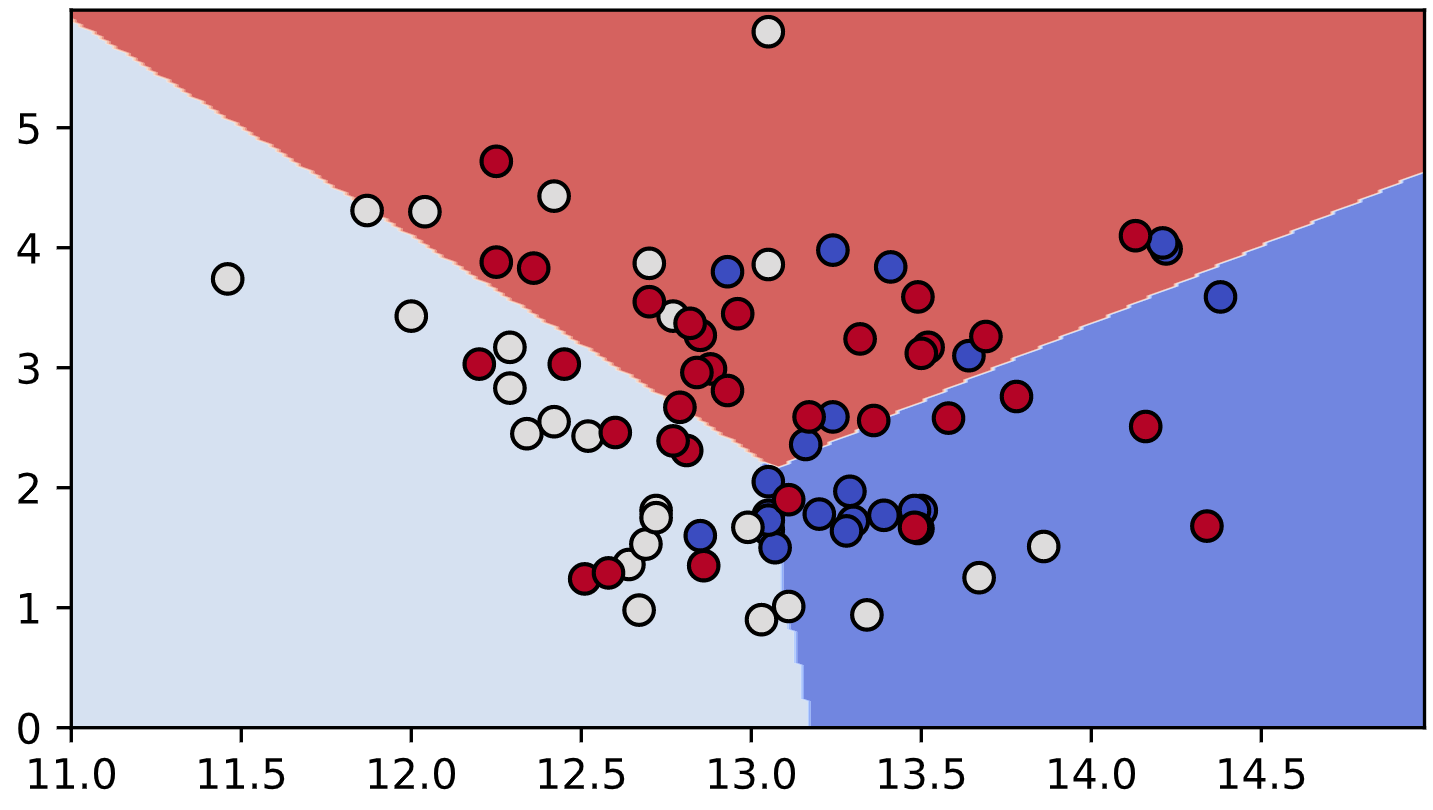
# Train a new SVM using only the support vectors

svm\_small = SVC(kernel="linear")

svm\_small.fit(X, y)

plot\_classifier(X\_small, y\_small, svm\_small, lims=(11,15,0,6))





In [1]: X, y

Out[1]:

(array([[14.23, 1.71],

[13.2 , 1.78],

[13.16, 2.36],

[14.37, 1.95],

[13.24, 2.59],

[14.2 , 1.76],

[14.39, 1.87],

[14.06, 2.15],

[14.83, 1.64],

[13.86, 1.35],

[14.1 , 2.16],

[14.12, 1.48],

[13.75, 1.73],

[14.75, 1.73],

[14.38, 1.87],

[13.63, 1.81],

[14.3 , 1.92],

[13.83, 1.57],

[14.19, 1.59],

[13.64, 3.1 ],

[14.06, 1.63],

[12.93, 3.8 ],

[13.71, 1.86],

[12.85, 1.6 ],

[13.5 , 1.81],

[13.05, 2.05],

[13.39, 1.77],

[13.3 , 1.72],

[13.87, 1.9 ],

[14.02, 1.68],

[13.73, 1.5 ],

[13.58, 1.66],

[13.68, 1.83],

[13.76, 1.53],

[13.51, 1.8 ],

[13.48, 1.81],

[13.28, 1.64],

[13.05, 1.65],

[13.07, 1.5 ],

[14.22, 3.99],

[13.56, 1.71],

[13.41, 3.84],

[13.88, 1.89],

[13.24, 3.98],

[13.05, 1.77],

[14.21, 4.04],

[14.38, 3.59],

[13.9 , 1.68],

[14.1 , 2.02],

[13.94, 1.73],

[13.05, 1.73],

[13.83, 1.65],

[13.82, 1.75],

[13.77, 1.9 ],

[13.74, 1.67],

[13.56, 1.73],

[14.22, 1.7 ],

[13.29, 1.97],

[13.72, 1.43],

[12.37, 0.94],

[12.33, 1.1 ],

[12.64, 1.36],

[13.67, 1.25],

[12.37, 1.13],

[12.17, 1.45],

[12.37, 1.21],

[13.11, 1.01],

[12.37, 1.17],

[13.34, 0.94],

[12.21, 1.19],

[12.29, 1.61],

[13.86, 1.51],

[13.49, 1.66],

[12.99, 1.67],

[11.96, 1.09],

[11.66, 1.88],

[13.03, 0.9 ],

[11.84, 2.89],

[12.33, 0.99],

[12.7 , 3.87],

[12. , 0.92],

[12.72, 1.81],

[12.08, 1.13],

[13.05, 3.86],

[11.84, 0.89],

[12.67, 0.98],

[12.16, 1.61],

[11.65, 1.67],

[11.64, 2.06],

[12.08, 1.33],

[12.08, 1.83],

[12. , 1.51],

[12.69, 1.53],

[12.29, 2.83],

[11.62, 1.99],

[12.47, 1.52],

[11.81, 2.12],

[12.29, 1.41],

[12.37, 1.07],

[12.29, 3.17],

[12.08, 2.08],

[12.6 , 1.34],

[12.34, 2.45],

[11.82, 1.72],

[12.51, 1.73],

[12.42, 2.55],

[12.25, 1.73],

[12.72, 1.75],

[12.22, 1.29],

[11.61, 1.35],

[11.46, 3.74],

[12.52, 2.43],

[11.76, 2.68],

[11.41, 0.74],

[12.08, 1.39],

[11.03, 1.51],

[11.82, 1.47],

[12.42, 1.61],

[12.77, 3.43],

[12. , 3.43],

[11.45, 2.4 ],

[11.56, 2.05],

[12.42, 4.43],

[13.05, 5.8 ],

[11.87, 4.31],

[12.07, 2.16],

[12.43, 1.53],

[11.79, 2.13],

[12.37, 1.63],

[12.04, 4.3 ],

[12.86, 1.35],

[12.88, 2.99],

[12.81, 2.31],

[12.7 , 3.55],

[12.51, 1.24],

[12.6 , 2.46],

[12.25, 4.72],

[12.53, 5.51],

[13.49, 3.59],

[12.84, 2.96],

[12.93, 2.81],

[13.36, 2.56],

[13.52, 3.17],

[13.62, 4.95],

[12.25, 3.88],

[13.16, 3.57],

[13.88, 5.04],

[12.87, 4.61],

[13.32, 3.24],

[13.08, 3.9 ],

[13.5 , 3.12],

[12.79, 2.67],

[13.11, 1.9 ],

[13.23, 3.3 ],

[12.58, 1.29],

[13.17, 5.19],

[13.84, 4.12],

[12.45, 3.03],

[14.34, 1.68],

[13.48, 1.67],

[12.36, 3.83],

[13.69, 3.26],

[12.85, 3.27],

[12.96, 3.45],

[13.78, 2.76],

[13.73, 4.36],

[13.45, 3.7 ],

[12.82, 3.37],

[13.58, 2.58],

[13.4 , 4.6 ],

[12.2 , 3.03],

[12.77, 2.39],

[14.16, 2.51],

[13.71, 5.65],

[13.4 , 3.91],

[13.27, 4.28],

[13.17, 2.59],

[14.13, 4.1 ]]),

array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2,

2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,

2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,

2, 2]))

In [2]: X\_train

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

X\_train

NameError: name 'X\_train' is not defined

In [3]: y\_train

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

y\_train

NameError: name 'y\_train' is not defined

In [4]: 😎😉

UnicodeEncodeError: 'utf-8' codec can't encode characters in position 0-3: surrogates not allowed

In [5]: 😎

UnicodeEncodeError: 'utf-8' codec can't encode characters in position 0-1: surrogates not allowed

In [6]: '😎'

UnicodeEncodeError: 'utf-8' codec can't encode characters in position 1-2: surrogates not allowed

<script.py> output:

Number of original examples 178

Number of support vectors 81

In [7]:

+70 XP

Nice! Compare the decision boundaries of the two trained models: are they the same? By the definition of support vectors, they should be!

**Exercise**

**Exercise**

**GridSearchCV warm-up**

In the video we saw that increasing the RBF kernel hyperparameter gamma increases training accuracy. In this exercise we'll search for the gamma that maximizes cross-validation accuracy using scikit-learn's GridSearchCV. A binary version of the handwritten digits dataset, in which you're just trying to predict whether or not an image is a "2", is already loaded into the variables X and y.

**Instructions**

**100 XP**

* Create a GridSearchCV object.
* Call the fit() method to select the best value of gamma based on cross-validation accuracy.

[**Take Hint (-30 XP)**](javascript:void(0))

In [1]: # 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, y)

# Report the best parameters

print("Best CV params", searcher.best\_params\_)

Best CV params {'gamma': 0.001}

<script.py> output:

Best CV params {'gamma': 0.001}

In [2]:

+100 XP

Great job! Larger values of gamma are better for training accuracy, but cross-validation helped us find something different (and better!).

**Exercise**

**Exercise**

**Jointly tuning gamma and C with GridSearchCV**

In the previous exercise the best value of gamma was 0.001 using the default value of C, which is 1. In this exercise you'll search for the best combination of C and gamma using GridSearchCV.

As in the previous exercise, the 2-vs-not-2 digits dataset is already loaded, but this time it's split into the variables X\_train, y\_train, X\_test, and y\_test. Even though cross-validation already splits the training set into parts, it's often a good idea to hold out a separate test set to make sure the cross-validation results are sensible.

**Instructions**

**100 XP**

* Run GridSearchCV to find the best hyperparameters using the training set.
* Print the best values of the parameters.
* Print out the accuracy on the test set, which was not used during the cross-validation procedure.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code generated an error. Fix it and try again!

**Hint**

The GridSearchCV object is fit using the same syntax as other scikit-learn objects.

# 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))

<script.py> output:

Best CV params {'C': 10, 'gamma': 0.0001}

Best CV accuracy 0.9988864142538976

Traceback (most recent call last):

File "script.py", line 14, in <module>

print("Test accuracy of best grid search hypers:", searcher.score(searcher.best\_params\_))

File "script.py", line 438, in score

return score(self.best\_estimator\_, X, y)

File "script.py", line 244, in \_passthrough\_scorer

return estimator.score(\*args, \*\*kwargs)

File "script.py", line 349, in score

return accuracy\_score(y, self.predict(X), sample\_weight=sample\_weight)

File "script.py", line 548, in predict

y = super(BaseSVC, self).predict(X)

File "script.py", line 308, in predict

X = self.\_validate\_for\_predict(X)

File "script.py", line 439, in \_validate\_for\_predict

X = check\_array(X, accept\_sparse='csr', dtype=np.float64, order="C")

File "script.py", line 433, in check\_array

array = np.array(array, dtype=dtype, order=order, copy=copy)

TypeError: float() argument must be a string or a number, not 'dict'

<script.py> output:

Best CV params {'C': 10, 'gamma': 0.0001}

Best CV accuracy 0.9988864142538976

Test accuracy of best grid search hypers: 0.9988876529477196

In [1]:

+70 XP

You got it! Note that the best value of gamma, 0.0001, is different from the value of 0.001 that we got in the previous exercise, when we fixed C=1. Hyperparameters can affect each other!

**An advantage of SVMs**

Which of the following is an advantage of SVMs over logistic regression?

**Answer the question**

**50 XP**

**Possible Answers**

They naturally outputs meaningful probabilities.

press

1

They can be used with kernels.

press

2

They are computationally efficient with kernels.

press

3

They learn sigmoidal decision boundaries.

press

4

**Incorrect Submission**

Kernel SVMs can learn all sorts of decision boundaries, but there's nothing particularly sigmoidal going on here.

+50 XP

That's right! Having a limited number of support vectors makes kernel SVMs computationally efficient.

**An advantage of logistic regression**

Which of the following is an advantage of logistic regression over SVMs?

**Answer the question**

**50 XP**

**Possible Answers**

It naturally outputs meaningful probabilities.

press

1

It can be used with kernels.

press

2

It is computationally efficient with kernels.

press

3

It learns sigmoidal decision boundaries.

press

4

+50 XP

You got it!

**Exercise**

**Exercise**

**Using SGDClassifier**

In this final coding exercise, you'll do a hyperparameter search over the regularization type, regularization strength, and the loss (logistic regression vs. linear SVM) using SGDClassifier().

**Instructions**

**100 XP**

* Instantiate an SGDClassifier instance with random\_state=0.
* Search over the regularization strength, the hinge vs. log losses, and L1 vs. L2 regularization.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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':['log'], 'penalty':['hinge']}

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))

**Incorrect Submission**

Check your call of GridSearchCV(). Did you correctly specify the second argument? Expected something different.

[2x]

Awesome, thanks for your feedback!

[**Show Answer (-70 XP)**](javascript:void(0))

**Hint**

* To search over the two losses put "hinge" and "log" in the list, in order.
* To search over the two regularization types put "l1" and "l2" in the list. Recall that sklearn uses lowercase letters to represent the regularization types.

# 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'], 'penalty':["l1", "l2"]}

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))

Traceback (most recent call last):

File "script.py", line 8, in <module>

searcher.fit(X\_train, y\_train)

File "script.py", line 639, in fit

cv.split(X, y, groups)))

File "script.py", line 779, in \_\_call\_\_

while self.dispatch\_one\_batch(iterator):

File "script.py", line 625, in dispatch\_one\_batch

self.\_dispatch(tasks)

File "script.py", line 588, in \_dispatch

job = self.\_backend.apply\_async(batch, callback=cb)

File "script.py", line 111, in apply\_async

result = ImmediateResult(func)

File "script.py", line 332, in \_\_init\_\_

self.results = batch()

File "script.py", line 131, in \_\_call\_\_

return [func(\*args, \*\*kwargs) for func, args, kwargs in self.items]

File "script.py", line 131, in <listcomp>

return [func(\*args, \*\*kwargs) for func, args, kwargs in self.items]

File "script.py", line 444, in \_fit\_and\_score

estimator.set\_params(\*\*parameters)

File "script.py", line 78, in set\_params

self.\_validate\_params(set\_max\_iter=False)

File "script.py", line 104, in \_validate\_params

self.\_get\_penalty\_type(self.penalty)

File "script.py", line 159, in \_get\_penalty\_type

raise ValueError("Penalty %s is not supported. " % penalty)

ValueError: Penalty hinge is not supported.

<script.py> output:

Best CV params {'alpha': 0.0001, 'loss': 'hinge', 'penalty': 'l1'}

Best CV accuracy 0.94351630867144

Test accuracy of best grid search hypers: 0.9592592592592593

<script.py> output:

Best CV params {'alpha': 0.0001, 'loss': 'hinge', 'penalty': 'l1'}

Best CV accuracy 0.94351630867144

Test accuracy of best grid search hypers: 0.9592592592592593

In [1]:

+70 XP

Congrats, you finished the last exercise in the course! One advantage of SGDClassifier is that it's very fast - this would have taken a lot longer with LogisticRegression or LinearSVC.