q1_q2_q3

March 25, 2021

[1]:
```
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score
from numpy import loadtxt, stack, mean, array, append, arange, absolute,␣
↪random, logical_and
import matplotlib.pyplot as plt
```

Q1 Part A: Build and SGDC Classifier and Report its Accuracy

[2]:
```
# import the data using numpy
var1, var2, y = loadtxt("./blobData.txt", delimiter=",", unpack=True)
X = stack([var1, var2], axis=1)
```

[3]:
```
# create the classifier and fit it to the data
classifier = make_pipeline(StandardScaler(), SGDClassifier())
classifier.fit(X, y)
```

[3]:
```
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('sgdclassifier', SGDClassifier())])
```

[4]:
```
classification_score = classifier.score(X, y)
print("Classification Score (Linear Features): {}".format(classification_score))
```

```
Classification Score (Linear Features): 0.34
```

Q1 Part B: Extract Quadratic Features and Report on Classificiation Accurracy

[5]:
```
# generate the polynomial features
poly = PolynomialFeatures(2)
quadratic_features = poly.fit_transform(X)
```

[6]:
```
# create the classifier and fit it to the data
classifier_quadratic = make_pipeline(StandardScaler(), SGDClassifier())
classifier_quadratic.fit(quadratic_features, y)
```

[6]:
```
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('sgdclassifier', SGDClassifier())])
```

```
[7]: classification_score_quadratic = classifier_quadratic.score(quadratic_features,␣
      ↪y)
     print("Classification Score (Quadratic Features): {}".
      ↪format(classification_score_quadratic))
```

```
Classification Score (Quadratic Features): 0.96
```

The quadratic feature extraction seems to be extermely benificial. Before quadratic feature extraction, the average accuracy on the test data is around 50%, however, the model fitted with the quadratic features has an average accuracy of 95%, a significant jump from the 50% average of the linear features. Because of this, the quadratic features are a significant improvement.

Q2 Part A: Use Cross Validation to Test Performance on Unseen Samples

```
[8]: # get the score for the original fit
     scores = cross_val_score(classifier, X, y, cv=8)

     print("Mean Cross Validation Score (Linear): {}".format(mean(scores)))
```

```
Mean Cross Validation Score (Linear): 0.5929487179487178
```

```
[9]: # get the score for the quadratic fit
     scores_quadratic = cross_val_score(classifier_quadratic, quadratic_features, y,␣
      ↪cv=8)

     print("Mean Cross Validation Score (Quadratic): {}".
      ↪format(mean(scores_quadratic)))
```

```
Mean Cross Validation Score (Quadratic): 0.9391025641025641
```

Q2 Part B: Report Cross Validations Averages Over Polynomials of Varying Degrees

```
[10]: for degree in [2, 4, 6]:

          # generate the polynomial features
          poly = PolynomialFeatures(degree)
          features = poly.fit_transform(X)

          # create the classifier and fit it to the data
          classifier = make_pipeline(StandardScaler(), SGDClassifier())
          classifier.fit(features, y)

          score_sum = 0
          # run the cross validation 1000 times
          for _ in range(1000):

              # get the score for the quadratic fit
              scores = cross_val_score(classifier, features, y, cv=8)
              score_sum += mean(scores)
```

```
    avg_total = score_sum / 1000
    print("Mean Cross Validation Score ({} Degrees): {}".format(degree,␣
↪avg_total))
```

```
Mean Cross Validation Score (2 Degrees): 0.9288068910256416
Mean Cross Validation Score (4 Degrees): 0.920877403846155
Mean Cross Validation Score (6 Degrees): 0.9298020833333341
```

Q3 Part A: Compute Mean Values of Coefficients over 1000 Iterations

[11]:
```
# repeat the SGDClassifier fitting over degree 2 data 1000 times

# generate the polynomial features
poly = PolynomialFeatures(2, include_bias=False)
features = poly.fit_transform(X)

# create the classifier and fit it to the data
classifier = make_pipeline(SGDClassifier())

# hold the coefficients and intercepts from each iteration
coeffs = []
intercepts = []

for _ in range(1000):

    classifier.fit(features, y)
    curr_coeffs = classifier.named_steps['sgdclassifier'].coef_
    curr_intercept = classifier.named_steps['sgdclassifier'].intercept_

    coeffs.append(curr_coeffs[0])
    intercepts.append(curr_intercept[0])

# typecast coeffs and intercept to a numpy array
coeffs = array(coeffs)
intercepts = array(intercepts)

coeff_averages = []

# add the average of the intercepts to the coefficient average
coeff_averages.append(mean(intercepts))

# iterate through the coefficients number
for coeff in range(0, coeffs.shape[1]):

    mean_coeff = mean(coeffs[:, coeff])

    coeff_averages.append(mean_coeff)
```

```
print("Averages", coeff_averages)
```

Averages [-390.65599377785026, 2.0570894064026373, -2.5182348034438724, 63.69063999782291, 2.791762916805224, 62.34987166608395]

[12]:
```
print("(mean) Intercept: {}".format(coeff_averages[0]))

for i, val in enumerate(coeff_averages):

    # skip the bias term
    if i == 0:
        continue

    print("(mean) Coefficient {}: {}".format(i, val))
```

```
(mean) Intercept: -390.65599377785026
(mean) Coefficient 1: 2.0570894064026373
(mean) Coefficient 2: -2.5182348034438724
(mean) Coefficient 3: 63.69063999782291
(mean) Coefficient 4: 2.791762916805224
(mean) Coefficient 5: 62.34987166608395
```

Q3 Part B: Examining the Coefficients from the Previous Part

[13]:
```
# get the maximum value of the coefficients
max_coeff = max(coeff_averages)

# set a threshold value for the coefficients we ignore
thresh_val = .05 * abs(max_coeff)
print(thresh_val)
```

3.1845319998911457

[14]:
```
coeffs_pass_threshold = absolute(coeff_averages) > thresh_val
coeffs_pass_threshold
```

[14]:
```
array([ True, False, False,  True, False,  True])
```

[15]:
```
# determine what variables are associated with the remaining coefficients
# get the powers
poly = PolynomialFeatures(2)

# fit another PolynomialFeatures object, this time including the bias so we can␣
↪get correct feature names
poly.fit_transform(X)
print(poly.powers_)
kept_powers = poly.powers_[coeffs_pass_threshold]
kept_powers
```

```
[[0 0]
 [1 0]
 [0 1]
 [2 0]
 [1 1]
 [0 2]]
```

[15]:
```
array([[0, 0],
       [2, 0],
       [0, 2]], dtype=int64)
```

[16]:
```
# check that the powers are consistent by getting variable names
kept_names = array(poly.get_feature_names())[coeffs_pass_threshold]
kept_names
```

[16]:
```
array(['1', 'x0^2', 'x1^2'], dtype='<U5')
```

[17]:
```
# what are the values of the coefficients we keep
# this array includes the intercept
coeffs_kept = array(coeff_averages)[coeffs_pass_threshold]
coeffs_kept
```

[17]:
```
array([-390.65599378,   63.69064   ,   62.34987167])
```

The equation for the decision boundary is given by

$$390.6068 = (63.4984)X_1^2 + (62.4624)X_2^2$$

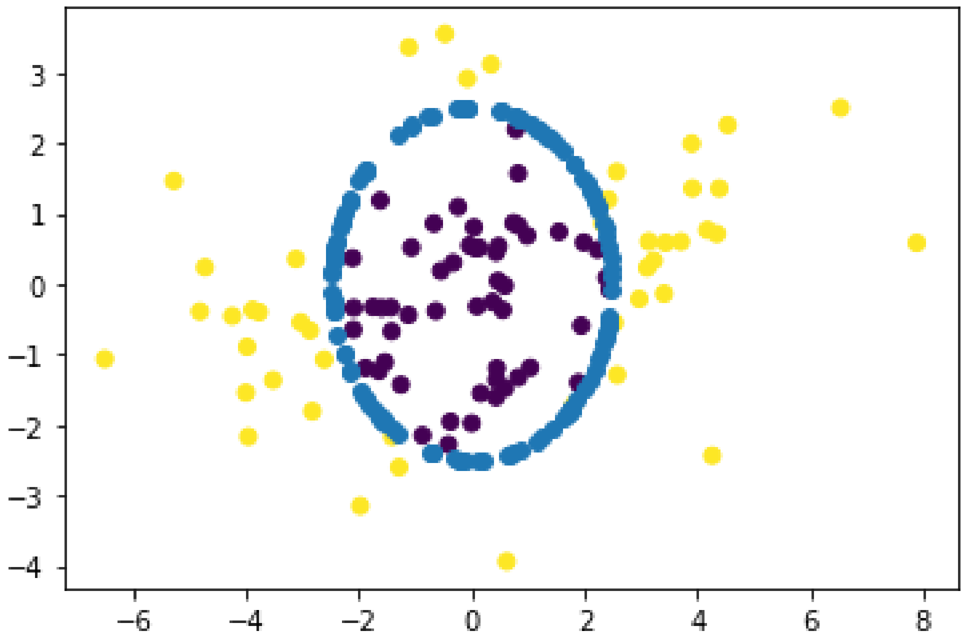
Based on this equation, we see that the shape of the decision boundary is a circle (not a perfect circle but the coefficients are large enough and close enough that it essentially is one). As we see in the attribute space plot below, the boundary between the two classes is essentially a circular, so this equation for the decision boundary makes sense.

[18]:
```
plt.scatter(var1, var2, c=y)

p = random.uniform(-3, 3, size=(1000000, 2))
c = array([63.49840257, 62.46239314])

points = p[logical_and((p**2).dot(c) - 390.60683054 > 0, (p**2).dot(c) - 390.
 ↪6068 < .1)]
plt.scatter(points[:, 0], points[:, 1])
```

[18]:
```
<matplotlib.collections.PathCollection at 0x1dbd12d6a00>
```



3
2
1
0
−1
−2
−3
−4
−6
−4
−2
0
2
4
6
8

q4.py

March 25, 2021

[1]:
```
from random import shuffle
from math import sqrt
```

Part A: Use Monte Carlo Simulation to Estimate P(5 examples in training set)

[2]:
```
num_iter = 10**3
indexes = list(range(0, 100))

conditions_met = []
for _ in range(num_iter):

    # create a list of indexes and shuffle it
    shuffle(indexes)

    training_indices = indexes[0:75]
    testing_indices = indexes[75:100]

    conditions_met.append(all(x in training_indices for x in [0, 1, 2, 3, 4]))

estimated_probability = sum(conditions_met) / num_iter
print("Estimated Probability of having special examples in our training split:␣
 ↪{}".format(estimated_probability))
```

```
Estimated Probability of having special examples in our training split: 0.226
```

Part B: Provide an Estimate of the Magnitude of the Error

There is a 95% chance that the actual probability will we within a distance of

$$1/\sqrt{N}$$

from the estimated value

[3]:
```
# calculate error using the formula above
error = 1 / sqrt(num_iter)
CI_95_percent = [round(estimated_probability - error, 3),␣
 ↪round(estimated_probability + error, 3)]
```

```
print("A 95% Confidence Interval for the probability is given by: {}".
→format(CI_95_percent))
```

```
A 95% Confidence Interval for the probability is given by: [0.194, 0.258]
```

Part C: Compute Monte Carlo Estimate of Number of Repititions For Which We Can be Sure that the error is accurate to three digits

We want to be sure that our estimate is accurate to the third decimal place, so we can set up the following inequality to represent this (noting that .0005 rounds up to .001):

$$\frac{1}{\sqrt{N}} \leq .0005$$

Analytically, we see that N must be at least 4,000,000 for us to be sure we're accurate within three decimal places. We check below using Monte Carlo simulation.

[4]:
```
N = 1

while True:

    error = 1 / sqrt(N)

    if error <= .0005:
        print("Lowest N: {}".format(N))
        break

    N += 1
```

```
Lowest N: 4000000
```

[5]:
```
# finally, compute the final estimated probability value
# same process from above but with our new lowest value of N s.t. we can be␣
→sure the estimate will be within 3 decimal place

conditions_met = []
indexes = list(range(0, 100))

for _ in range(N):

    # create a list of indexes and shuffle it
    shuffle(indexes)

    training_indices = indexes[0:75]
    testing_indices = indexes[75:100]

    conditions_met.append(all(x in training_indices for x in [0, 1, 2, 3, 4]))
```

```
estimated_probability = sum(conditions_met) / N
print("Estimated Probability of having special examples in our training split: {}".format(estimated_probability))
```

```
Estimated Probability of having special examples in our training split:
0.22912625
```