lab06_student_version

October 20, 2021

1 Lab 6: Nonparametric methods

Welcome to the sixth DS102 lab!

The goal of this lab is to explore and interpret several nonparametric methods for regression.

The code you need to write is commented out with a message "TODO: fill in". There is additional documentation for each part as you go along.

In preparation for this lab we would recommend that you review the slides and demos from Lectures 14 and 15.

Collaboration Policy

Data science is a collaborative activity. While you may talk with others about the labs, we ask that you **write your solutions individually**. If you do discuss the assignments with others please **include their names** in the cell below.

Gradescope Submission

To submit this assignment, rerun the notebook from scratch (by selecting Kernel > Restart & Run all), and then print as a pdf (File > download as > pdf) and submit it to Gradescope.

This assignment should be completed and submitted before Wednesday, October 20, 2021 at 11:59 PM. PST

Collaborators

Write the names of your collaborators in this cell.

<Collaborator Name> <Collaborator e-mail>

[1]:
```
import numpy as np
import pandas as pd
pd.options.mode.chained_assignment = None

%matplotlib inline

import matplotlib.pyplot as plt
import matplotlib.patches as patches
import seaborn as sns
```

```
sns.set()

import hashlib
def get_hash(num, significance = 4):
    num = round(num, significance)
    """Helper function for assessing correctness"""
    return hashlib.md5(str(num).encode()).hexdigest()
```

2 1. Model comparison

In this lab, we'll be working with the hybrid car dataset (which you may remember from Data 8).

It contains data on 153 different models of hybrid car from 1997 to 2013, with the price (`msrp`), gas mileage (`mpg`), type of car (`class`), and how fast the car accelerates in km/hour/second (`acceleration`).

We're going to try to predict the price using other features of the car.

[2]:
```
hybrid = pd.read_csv('hybrid.csv')

X_cols = ["year", "acceleration", "mpg"] # Columns used for prediction
y_col = "msrp" # The column we're trying to predict

hybrid
```

[2]:

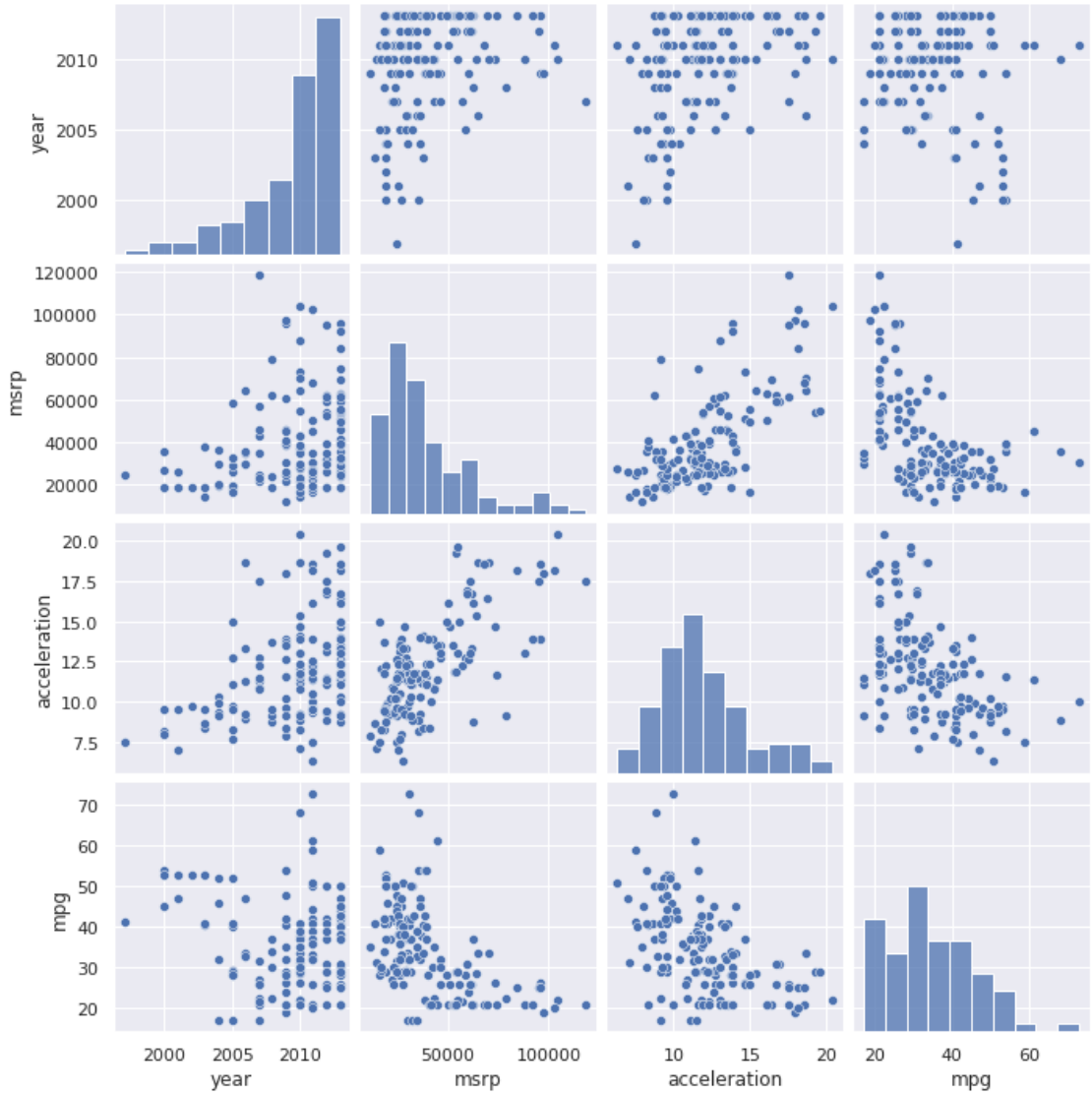
	vehicle	year	msrp	acceleration	mpg	class
0	Prius (1st Gen)	1997	24509.74	7.46	41.26	Compact
1	Tino	2000	35354.97	8.20	54.10	Compact
2	Prius (2nd Gen)	2000	26832.25	7.97	45.23	Compact
3	Insight	2000	18936.41	9.52	53.00	Two Seater
4	Civic (1st Gen)	2001	25833.38	7.04	47.04	Compact
..	…	…	…	…	…	…
148	S400	2013	92350.00	13.89	21.00	Large
149	Prius Plug-in	2013	32000.00	9.17	50.00	Midsize
150	C-Max Energi Plug-in	2013	32950.00	11.76	43.00	Midsize
151	Fusion Energi Plug-in	2013	38700.00	11.76	43.00	Midsize
152	Chevrolet Volt	2013	39145.00	11.11	37.00	Compact

[153 rows x 6 columns]

This cell generates all pairs of scatterplots for numerical variables in the data. You should see the same trends discussed in the chapter of the Data 8 textbook linked above.

[3]:
```
sns.pairplot(hybrid)
```

[3]:
```
<seaborn.axisgrid.PairGrid at 0x7f5fc40964c0>
```



2.0.1 1(a) Splitting the data

We'll start by splitting the data into training and test sets. Use the scikit-learn function `train_test_split` to make two dataframes called `train` and `test`. The test set should have 30% of the data (46 rows).

The `train_test_split` function has an argument called `random_state` that lets you ensure that it uses the same random split every time: you should set that argument to `102` to pass the tests.

[4]:
```
# TODO: fill in
from sklearn.model_selection import train_test_split

train, test = train_test_split(hybrid, test_size=0.30, random_state=102) # TODO:
 ↪ fill in
```

[5]:
```
# Validation Test
assert(get_hash(train) == 'a8821bb31a8ae0d2b7803004be63656d')
assert(get_hash(test) == '96dbf091169d3503cd962bbba32db3f3')
print("Test passed!")
```

```
Test passed!
```

2.0.2 1(b) Predicting the output

1.b.(i) Linear regression Use linear regression to predict the MSRP from year, acceleration, and MPG. Add a new column to the `train` and `test` dataframes called `linear_pred` with the predictions from linear regression.

Hint: throughout this lab, you should use the default values of all parameters for all models we're experimenting with.

Hint: for this lab, you don't need to worry about pandas warnings about setting a value on a copy of a slice.

[6]:
```
from sklearn.linear_model import LinearRegression

linear_model = LinearRegression()
X = train[X_cols]
y = train[y_col]

linear_model = LinearRegression().fit(X, y)

train["linear_pred"] = linear_model.predict(X) # TODO: fill in
test["linear_pred"] = linear_model.predict(test[X_cols]) # TODO: fill in
```

[7]:
```
# Validation Test
assert(get_hash(train["linear_pred"]) == '42969b6301fad9e5e98680924754e9d6')
assert(get_hash(test["linear_pred"]) == '104e14d773c8efc0099711f9e62f0123')
assert(isinstance(linear_model, LinearRegression))
print("Test passed!")
```

```
Test passed!
```

Run the following cell that computes the training set error and test set error.

[8]:
```
train_rmse = np.mean((train["linear_pred"] - train["msrp"]) ** 2) ** 0.5
test_rmse = np.mean((test["linear_pred"] - test["msrp"]) ** 2) ** 0.5

print("Training set error for linear model:", train_rmse)
print("Test set error for linear model:    ", test_rmse)
```

```
Training set error for linear model: 15139.208226548888
Test set error for linear model:     13521.818567754557
```

1.b.(ii) Decision Trees Recall that a decision tree is a method for classification and regression that uses a tree-like structure to decide what value to predict for a point.

In this question, we'll use a decision tree for regression instead of classification. When we built a decision tree for classification in lecture, we made decisions about splitting based on how homogeneous the y-values were. Now, we'll instead make splits based on the residuals for predicting at that node.

Let's look at an example, assuming we're using mean squared error as our loss. For example, if we make our first split based on whether or not `mpg <= M`, we'll have some average MSRP for the low-MPG cars (below `M`), along with residuals if we used that average to predict the MSRP for all the low-MPG cars. Similarly, we have the same information for the high-MPG cars (above `M`). A good value of `M` will make the mean squared residuals for the two groups as small as possible. So, at each node, we choose a split that makes the mean squared error on each side as small as possible.

Compute the prediction from a decision tree with the default parameters for scikit-learn's `DecisionTreeRegressor` (i.e., no limit on tree depth). Add a new column to the `train` and `test` dataframes called `tree_pred` with the predictions from the decision tree.

Hint: your code should look very similar to your answer from 1.b.(i).

[9]:
```
from sklearn.tree import DecisionTreeRegressor

tree_model = DecisionTreeRegressor().fit(X, y) # TODO: fill in

train["tree_pred"] = tree_model.predict(X) # TODO: fill in
test["tree_pred"] = tree_model.predict(test[X_cols]) # TODO: fill in
```

[10]:
```
# Validation Test
assert(get_hash(train["tree_pred"]) == 'dbcb9fedf8018825bf704351b0cd7f9d')
assert(isinstance(tree_model, DecisionTreeRegressor))
print("Test passed!")
```

```
Test passed!
```

Run the following cell that computes the training set error and test set error.

[11]:
```
train_rmse = np.mean((train["tree_pred"] - train["msrp"]) ** 2) ** 0.5
test_rmse = np.mean((test["tree_pred"] - test["msrp"]) ** 2) ** 0.5

print("Training set error for decision tree:", train_rmse)
print("Test set error for decision tree:    ", test_rmse)
```

```
Training set error for decision tree: 393.0619080391814
Test set error for decision tree:     16303.938013469186
```

1.b.(iii) Random Forest Recall that a random forest is the combination of a large number of decision trees.

Compute the prediction from a decision tree using scikit-learn's `RandomForestRegressor`, with the following parameters: * 100 trees (default) * no limit on each tree's depth (default) * Use

the `max_features` parameter to only use one feature for each tree. (*The recommended value for random forests is for each tree to only use 1/3 of the features, and in this case we have 3 features.*)

Add a new column to the `train` and `test` dataframes called `forest_pred` with the predictions from the random forest.

Hint: your code should look very similar to your answers from 1.b.(i) and 1.b.(ii).

[12]:
```
from sklearn.ensemble import RandomForestRegressor

forest_model = RandomForestRegressor(max_features = 1).fit(X, y) # TODO: fill in


train["forest_pred"] = forest_model.predict(X) # TODO: fill in
test["forest_pred"] = forest_model.predict(test[X_cols]) # TODO: fill in
```

[13]:
```
# Validation Test: If you aren't passing these, your random forest is doing␣
 ↪something wrong
assert((((train["forest_pred"] - train["msrp"]) **2).sum()) < 3981470000.0)
assert((((test["forest_pred"] - test["msrp"]) **2).sum()) < 8015040000.0)
assert(isinstance(forest_model, RandomForestRegressor))
assert(forest_model.max_features != 'auto')
print("Test passed!")
```

```
Test passed!
```

Run the following cell that computes the training set error and test set error.

[14]:
```
train_rmse = np.mean((train["forest_pred"] - train["msrp"]) ** 2) ** 0.5
test_rmse = np.mean((test["forest_pred"] - test["msrp"]) ** 2) ** 0.5

print("Training set error for random forest:", train_rmse)
print("Test set error for random forest:    ", test_rmse)
```

```
Training set error for random forest: 5429.1572119884395
Test set error for random forest:     11881.354330825763
```

2.0.3 1(c) accuracy comparison

Of the Decision Tree model and the Random Forest model, which one does best on the training set? Why? Which model does best on the test set? Why?

The Decision Tree model does best on the training set because decision trees achieve very very high accuracy. They do this by overfitting on the training set, and spending more time so they can accurately classify each point. As a result, the Random Forest model does best on the test set because the Random Forest model has lower variance because it fits the training data less closely, and avoids the problem of overfitting.

2.0.4 1.d Interpretability

1.d.i Linear Regression Let's look at the coefficients from the linear regression model:

[15]:
```
# You can just run this cell to print out the coefficients for each feature:
print(X_cols)
linear_model.coef_
```

```
['year', 'acceleration', 'mpg']
```

[15]:
```
array([-255.24645465, 4260.68613645, -458.41724788])
```

Using this result, fill in the blanks in the following two statements:

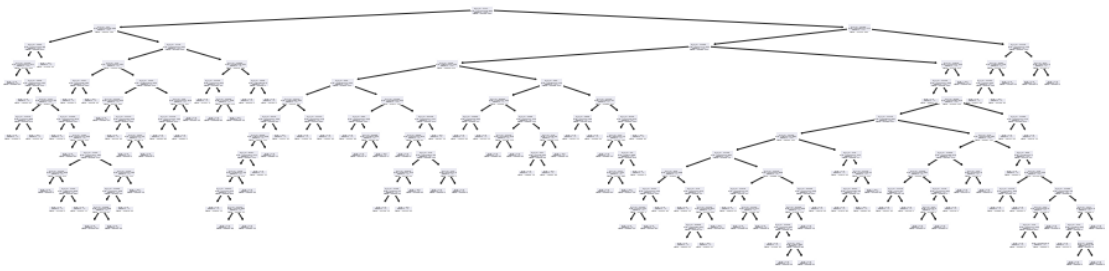
"Each year, linear regression predicts that the average price changes by $_____". (Your answer should be a positive or negative number)

"Linear regression predicts that cars with better gas mileage are _____ expensive." (Your answer should be either 'more' or 'less')

1. -255
2. less

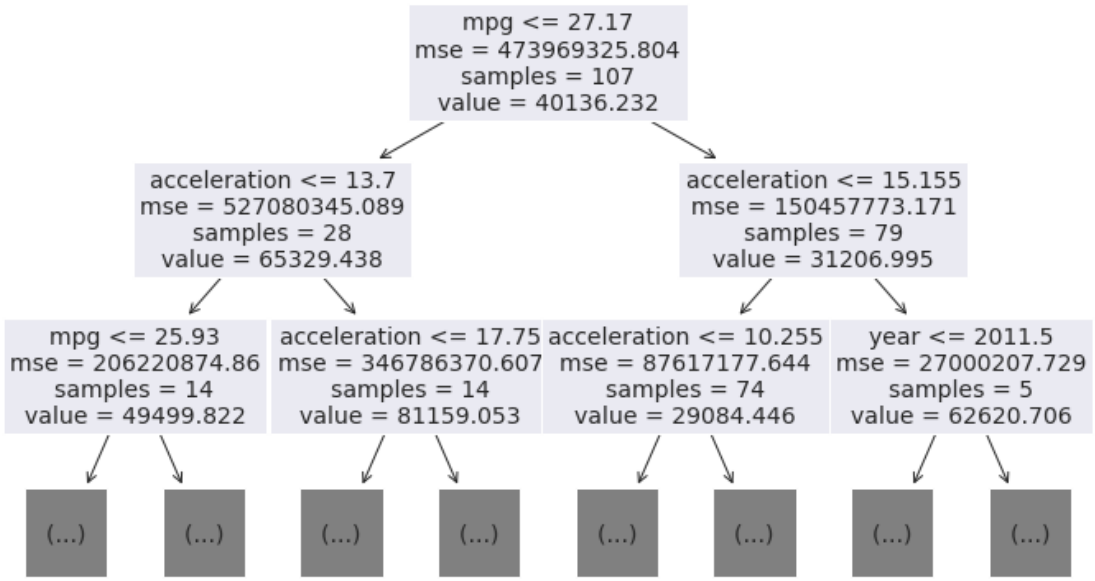
1.d.ii Decision trees We'll use the `plot_tree` function to draw the decision tree:

[16]:
```
from sklearn.tree import plot_tree
plt.figure(figsize=(16, 4))
plot_tree(tree_model);
```



We can see that the tree is quite deep and complex. Let's take a closer look at the nodes at the top:

[17]:
```
plt.figure(figsize=(12, 7))
plot_tree(tree_model, max_depth=2, fontsize=14, feature_names=X_cols);
```



There are a few things we can see right away:

- The first line tells us which feature to split on: values below the threshold go to the left, and values above go to the right.
- The third line tells us the number of training samples that made it that far into the tree.
- The fourth line tells us the average y-value (in this case, MSRP) of all the training samples that made it that far into the tree.

Just by looking at the first few layers, we can already see that the decision tree has pulled out the most expensive cars into some of the branches, and the less expensive ones into other branches.

Suppose we had stopped growing the tree at this point. That would have given us four leaf nodes, each with very different mean MSRP. Describe the node that contains the most expensive cars in plain English.

The most expensive cars lie in the node at the bottom of the tree, 2nd from the left. The average MSRP for these most expensive cars is \$81,159. They have an MPG less than 27 and an acceleration above 13.7.

Random Forests Unfortunately, random forests are much harder to interpret than either of the other two methods that we've tried. In this case, with so few features, we might be able to look at the top of each tree and find similarities across most or all of the trees, but in high-dimensional problems, each tree should see a very different subset of features, and this becomes much harder.

3 2. Explanations

Many methods for explainable ML use the following setup to explain a specific prediction from a complex model:

1. Construct a simpler, easier-to-explain model (e.g., linear regression, decision tree, etc.) that behaves similarly to the complex model for data points near the specific point we're trying to explain.
2. Interpret the simpler model.

In this question, we'll try to see if we can come up with an explanation for the worst predictions from each model.

3.0.1 2.a Linear Regression

2.a.i Finding the worst predictions Find the two cars in the test set where linear regression does the worst (i.e., has the highest absolute error, not the squared error). Your answer should be a dataframe with two rows from `test`, one for each of the two cars. You can add extra columns to `test` if you wish.

[18]:
```
# Hint: There are a couple of ways to do this. One suggestion is to first␣
↪create new column with the
# absolute linear error and then sort the dataframe based on this absolute␣
↪linear error.

worst_linear_predicted_cars_df = abs(test['linear_pred']-test['msrp'])
worst_linear_predicted_cars_df = worst_linear_predicted_cars_df.
↪sort_values(ascending=False) # TODO: fill in
worst_linear_predicted_cars_df = worst_linear_predicted_cars_df[:2]
```

[19]:
```
# Validation Test
assert(get_hash(worst_linear_predicted_cars_df.index.values[0]) ==␣
↪'6ea9ab1baa0efb9e19094440c317e21b')
assert(get_hash(worst_linear_predicted_cars_df.index.values[1]) ==␣
↪'d09bf41544a3365a46c9077ebb5e35c3')
print("Test passed!")
```

```
Test passed!
```

2.a.ii Explanation Using the coefficients of the linear model that we found earlier, explain why linear regression's predictions for these two cars were the way they were. Is the explanation from the linear model consistent with the trends you observed at the beginning in the visualizations?

[20]:
```
test['mpg'].var()
```

[20]:
```
116.69572193236719
```

[21]:
```
test['acceleration'].var()
```

[21]:
```
8.08664811594203
```

[22]:
```
test['year'].var()
```

[22]: 12.29227053140096

TODO: Your answer here:

First part: Linear regression's predictions for these two cars were the way they were because of the negative coefficient next to year. Both these cars have a low MPG (21.0) compared to the mean MPG of ~33. Since MPG has such large variance compared to the other two variables, this ends up penalizing these two cars a lot more.

Second part: Previously the coefficients were the following:

year, acceleration, mpg = -255.24645465, 4260.68613645, -458.41724788.

This is consistent with the trends from the visualizations at the beginning as the relationship between mpg and mrsp is also a stronger negative one and the relationship between acceleration and mrsp is a very strong positive one, which matches conclusions drawn from the coefficients. However, the relationship between year and mrsp is a positive one which does not match our conclusion of a negative coefficient.

3.0.2 2.b: Random Forests

2.b.i Finding the worst predictions for the random forest Find the two cars in the test set where random forest does the worst (i.e., has the highest absolute error). Your answer should be a dataframe with two rows from `test`, one for each of the two cars. You can add extra columns to `test` if you wish.

Hint: your code should be very similar to your code for 2.a.i.

[23]:
```
worst_forest_predicted_cars_df = abs(test['forest_pred'] - test['msrp']) # TODO:
 ↪ fill in
worst_forest_predicted_cars_df = worst_forest_predicted_cars_df.
 ↪sort_values(ascending=False)
worst_forest_predicted_cars_df = worst_forest_predicted_cars_df[:2]
```

[24]:
```
# Validation Test
assert(get_hash(worst_forest_predicted_cars_df.index.values[0]) ==␣
 ↪'6ea9ab1baa0efb9e19094440c317e21b')
assert(get_hash(worst_forest_predicted_cars_df.index.values[1]) ==␣
 ↪'d09bf41544a3365a46c9077ebb5e35c3')
print("Test passed!")
```

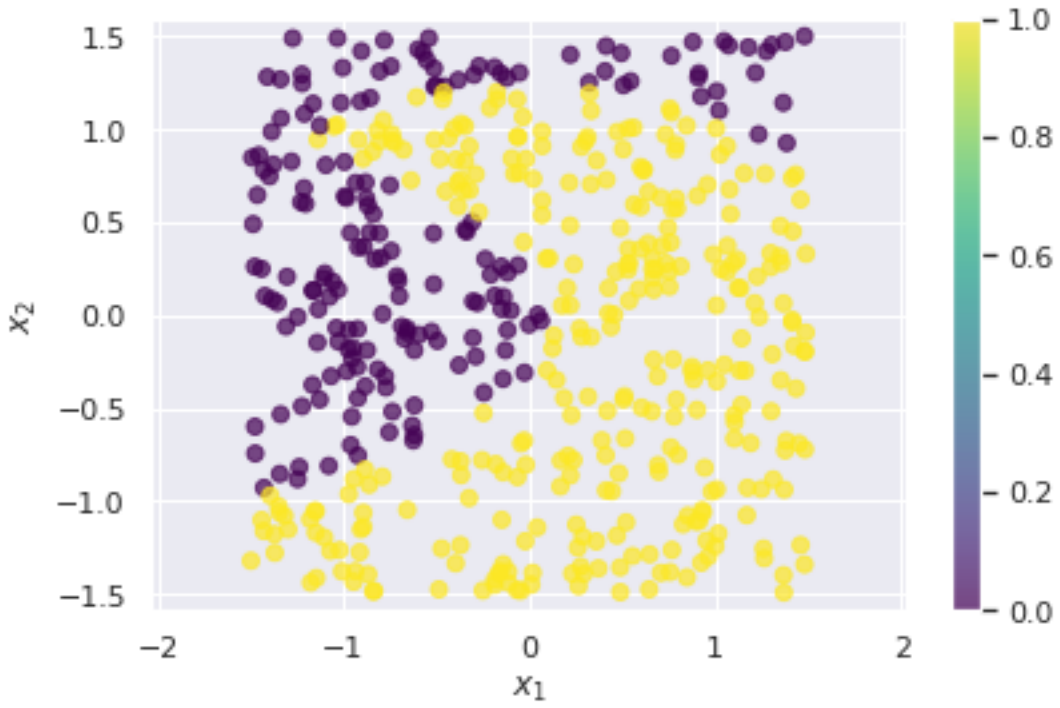
```
Test passed!
```

4 3 Feature Engineering and Interpretability

In this question, we will be exploring the effect of feature engineering on the interpretability of a given model using a toy dataset. Let's start by loading the data, which has already been split for you into train and test sets:

[25]:
```
# This is the same plotting function from lecture

def draw_results(x1, x2, color, plot_title=''):
    plt.figure()
    plt.scatter(x1, x2, c=color, cmap='viridis', alpha=0.7);
    plt.colorbar()
    plt.title(plot_title)
    plt.axis('equal')
    plt.xlabel('$x_1$')
    plt.ylabel('$x_2$')
    plt.tight_layout()
```

[26]:
```
# Import datasets
ring_train = pd.read_csv('ring_train.csv')
ring_test = pd.read_csv('ring_test.csv')
draw_results(ring_train['x1'], ring_train['x2'], color=ring_train['y'])
```



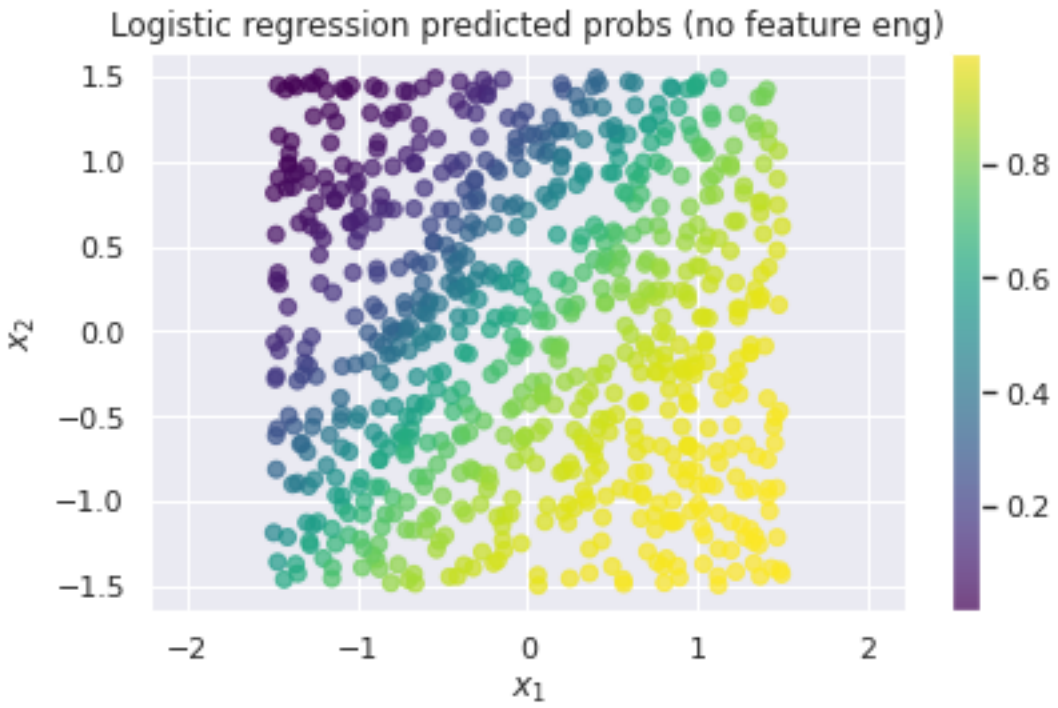
We know from lecture that without any additional features, logistic regression will use a line as a decision boundary. Where would you draw the best line to classify these points? (No need to answer, but please think about it.)

We are now going to fit a simple logistic regression model on the data using the following lines of code.

[27]:
```
# No need to write any code here: just understand.
X_train = ring_train[['x1', 'x2']].values
y_train = ring_train['y'].values

X_test = ring_test[['x1', 'x2']].values
y_test = ring_test['y'].values
```

[28]:
```
# No need to write any code here: just understand.
from sklearn.linear_model import LogisticRegression

X_train = ring_train[['x1', 'x2']].values
y_train = ring_train['y'].values

X_test = ring_test[['x1', 'x2']].values
y_test = ring_test['y'].values

model_simple_features = LogisticRegression(
    penalty='none', solver='lbfgs'
)

model_simple_features.fit(X_test, y_test)

probs = model_simple_features.predict_proba(X_test)[:, 1]
y_hat = (probs > 0.5).astype(np.int64)

draw_results(
    X_test[:, 0], X_test[:, 1], color=probs,
    plot_title="Logistic regression predicted probs (no feature eng)"
)
draw_results(
    X_test[:, 0], X_test[:, 1], color=y_hat,
    plot_title="Logistic regression prediction (no feature eng)"
)

accuracy = np.mean(y_test == y_hat)
print(f"Accuracy on training set: {accuracy}")
```

```
Accuracy on training set: 0.7933333333333333
```



Comparing the labels classified by the model with the true labels, we notice that the simple logistic

regression model is not ideal partly because the true decision boundary is nonlinear. In order to improve on the model, we will engineer new features.

With the checkerboard dataset, we engineered a new feature, $x_1 \times x_2$, which was just what we needed. For this dataset, the feature that we need is a little more complicated.

Instead, we'll take inspiration from neural networks, and add many random features, where each is a random linear combination of the inputs, where the coefficients will be random numbers between -1 and 1.

Don't forget that we also need to apply a nonlinearity, or else the linear combinations won't help us when applying logistic regression. In this example, we'll use the sigmoid function. For example, one feature might be $\sigma(-0.37x_1 + 0.82x_2)$.

4.0.1 3.a Add random features

Complete the cell below to add random features to the dataset. As described above, we first generate a pair of coefficients (c_1, c_2) uniformly random from $(-1, 1)$ and then for botht the training set and the test set, add an additional column whose values are $c_1x_1 + c_2x_2$.

[29]:
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def add_random_feature(train_data, test_data):
    # Returns the modified train_data and test_data
    coeffs = np.random.uniform(low=-1, high=1, size=2) # TODO: fill in
    # This code gives the feature a convenient name
    feat_name = f" ({coeffs[0]:0.2f}x1 + {coeffs[1]:0.2f}x2)"

    for dataset in (train_data, test_data):
        linear_combination = np.dot(dataset[['x1', 'x2']], coeffs)

        feature = sigmoid(linear_combination) # TODO: fill in
        dataset[feat_name] = feature
    return train_data, test_data
```

[30]:
```
# Tests
ring_train_copy3a, ring_test_copy3a = ring_train.copy(), ring_test.copy()
for i in range(23):
    add_random_feature(ring_train_copy3a, ring_test_copy3a)

assert((ring_train_copy3a.iloc[:, 4:].values < 1).all())
assert((ring_train_copy3a.iloc[:, 4:].values > 0 ).all())
assert((ring_test_copy3a.iloc[:, 4:].values < 1 ).all())
assert((ring_test_copy3a.iloc[:, 4:].values > 0 ).all())
assert(ring_train_copy3a.shape == (500, 26))
assert(ring_test_copy3a.shape == (750, 26))
```

```
print("tests passed!")
```

```
tests passed!
```

Using the code you completed in 3a, we can now add 10 random features to both the training set and the test set using the following code:

[31]:
```
# This cell uses the code you wrote to add 10 random features to the
# train and test sets.

ring_train_feats = ring_train.copy()
ring_test_feats = ring_test.copy()
for i in range(10):
    ring_train_feats, ring_test_feats = (
        add_random_feature(ring_train_feats, ring_test_feats)
    )
ring_train_feats.head()
```

[31]:

	y	x1	x2	(0.32x1 + 0.79x2)	(0.20x1 + -0.66x2)	\
0	0	-0.188203	1.331702	0.730336	0.284165	
1	1	-1.441836	-1.103109	0.208025	0.608256	
2	1	-0.028040	-0.675134	0.367110	0.609034	
3	1	1.008068	0.051442	0.589907	0.542657	
4	1	-1.104465	-1.197234	0.213571	0.639057	

	(-0.77x1 + 0.96x2)	(-0.09x1 + -0.65x2)	(-0.17x1 + -0.43x2)	\
0	0.805651	0.298375	0.368896	
1	0.512565	0.700199	0.670891	
2	0.348415	0.609283	0.572702	
3	0.326170	0.469543	0.452471	
4	0.425716	0.706882	0.667284	

	(-0.34x1 + 0.53x2)	(0.65x1 + -0.49x2)	(0.23x1 + -0.04x2)	\
0	0.682545	0.316846	0.474876	
1	0.478043	0.401183	0.427919	
2	0.414382	0.576735	0.505530	
3	0.421276	0.652329	0.558129	
4	0.437140	0.466075	0.448323	

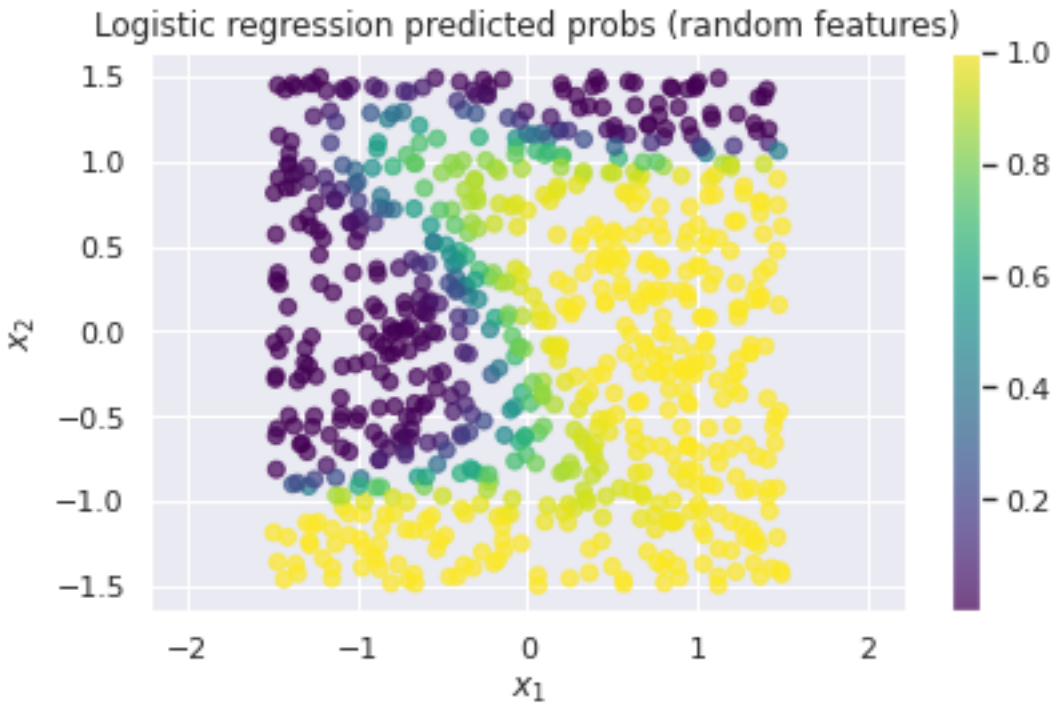
	(-0.66x1 + 0.52x2)	(-0.43x1 + -0.94x2)
0	0.693848	0.237659
1	0.594211	0.838956
2	0.417522	0.655619
3	0.344917	0.382175
4	0.527160	0.831167

We can now train a new logistic regression model with these 10 additional features.

[32]:
```
# No need to write any code here: just understand.
X_train = ring_train_feats.iloc[:, 1:].values
y_train = ring_train_feats['y'].values

X_test = ring_test_feats.iloc[:, 1:].values
y_test = ring_test_feats['y'].values

model_features = LogisticRegression(
    penalty='none', solver='lbfgs'
)

model_features.fit(X_train, y_train)

probs = model_features.predict_proba(X_test)[:, 1]
y_hat = (probs > 0.5).astype(np.int64)

draw_results(
    X_test[:, 0], X_test[:, 1], color=probs,
    plot_title="Logistic regression predicted probs (random features)"
)

draw_results(
    X_test[:, 0], X_test[:, 1], color=y_hat,
    plot_title="Logistic regression prediction (random features)"
)

accuracy = np.mean(y_test == y_hat)
print(f"Accuracy on training set: {accuracy}")
```

```
/opt/conda/lib/python3.9/site-packages/sklearn/linear_model/_logistic.py:763:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

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

```
Accuracy on training set: 0.9146666666666666
```



You should see that the accuracy is already better. Now that our model has improved, let's try to

interpret it.

4.0.2 3.b Coefficients

Fill in the blanks in the following code that creates a dataframe with the coefficients from the logistic regression model.

Hint: you may find it helpful to refer to the demo from lecture.

[33]:
```
# Complete the code below
feature_names = ring_train_feats.columns[1:]
logistic_coeff_vals = model_features.coef_[0] # TODO fill in
len(logistic_coeff_vals)
coefficient_df = pd.DataFrame(
    {'feature': feature_names, 'coefficients': logistic_coeff_vals}
)

coefficient_df
```

[33]:

	feature	coefficients
0	x1	47.522211
1	x2	-233.109632
2	(0.32x1 + 0.79x2)	298.056263
3	(0.20x1 + -0.66x2)	-467.661579
4	(-0.77x1 + 0.96x2)	-433.423721
5	(-0.09x1 + -0.65x2)	-297.999008
6	(-0.17x1 + -0.43x2)	58.764793
7	(-0.34x1 + 0.53x2)	413.766979
8	(0.65x1 + -0.49x2)	-238.735433
9	(0.23x1 + -0.04x2)	63.394788
10	(-0.66x1 + 0.52x2)	399.018497
11	(-0.43x1 + -0.94x2)	-115.739988

[34]:
```
# Tests
assert(coefficient_df.shape == (12, 2))
assert(coefficient_df['feature'][0] == 'x1')
assert(coefficient_df['feature'][1] == 'x2')

print("Tests passed!")
```

```
Tests passed!
```

4.0.3 3c Interpretibility

This model has better performance on the data but at the same time, it loses some interpretability. Explain why this logistic regression model is harder to interpret than the simpler (and worse-performing) one from earlier.

This one is harder to interpret because by wrapping the sigmoid function around x1 and x2, it's hard to say how exactly x1 and x2 themselves are affecting the MRSP. In the previous model, you

could directly read the coefficients off and see how big of an effect each feature has on the MRSP. Here, you cannot do this directly.

4.0.4 Congratulations on completing the lab!

[35]:
```
import matplotlib.image as mpimg
img = mpimg.imread('cute_flemish.jpg')
imgplot = plt.imshow(img)
imgplot.axes.get_xaxis().set_visible(False)
imgplot.axes.get_yaxis().set_visible(False)
print("Yay, you've made it to the end of Lab 9!")
plt.show()
```

```
Yay, you've made it to the end of Lab 9!
```