ML for everyone (chapter 4 - starting with Regression)

April 5, 2022

[1]:
```
# setup
from mlwpy import *
%matplotlib inline
```

1 A Simple Regression Dataset

Regression is the process of predicting a finely graded numerical value from inputs

[2]:
```
diabetes = datasets.load_diabetes()
diabetes_df = pd.DataFrame(diabetes.data, columns=diabetes.feature_names)
diabetes_df['target'] = diabetes.target
display(pd.concat([diabetes_df.head(3),diabetes_df.tail(3)]))
```

	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6	target
0	0.0381	0.0507	0.0617	0.0219	-0.0442	-0.0348	-0.0434	-0.0026	0.0199	-0.0176	151.0000
1	-0.0019	-0.0446	-0.0515	-0.0263	-0.0084	-0.0192	0.0744	-0.0395	-0.0683	-0.0922	75.0000
2	0.0853	0.0507	0.0445	-0.0057	-0.0456	-0.0342	-0.0324	-0.0026	0.0029	-0.0259	141.0000
439	0.0417	0.0507	-0.0159	0.0173	-0.0373	-0.0138	-0.0250	-0.0111	-0.0469	0.0155	132.0000
440	-0.0455	-0.0446	0.0391	0.0012	0.0163	0.0153	-0.0287	0.0266	0.0445	-0.0259	220.0000
441	-0.0455	-0.0446	-0.0730	-0.0814	0.0837	0.0278	0.1738	-0.0395	-0.0042	0.0031	57.0000

[4]:
```
diabetes_df.shape
```

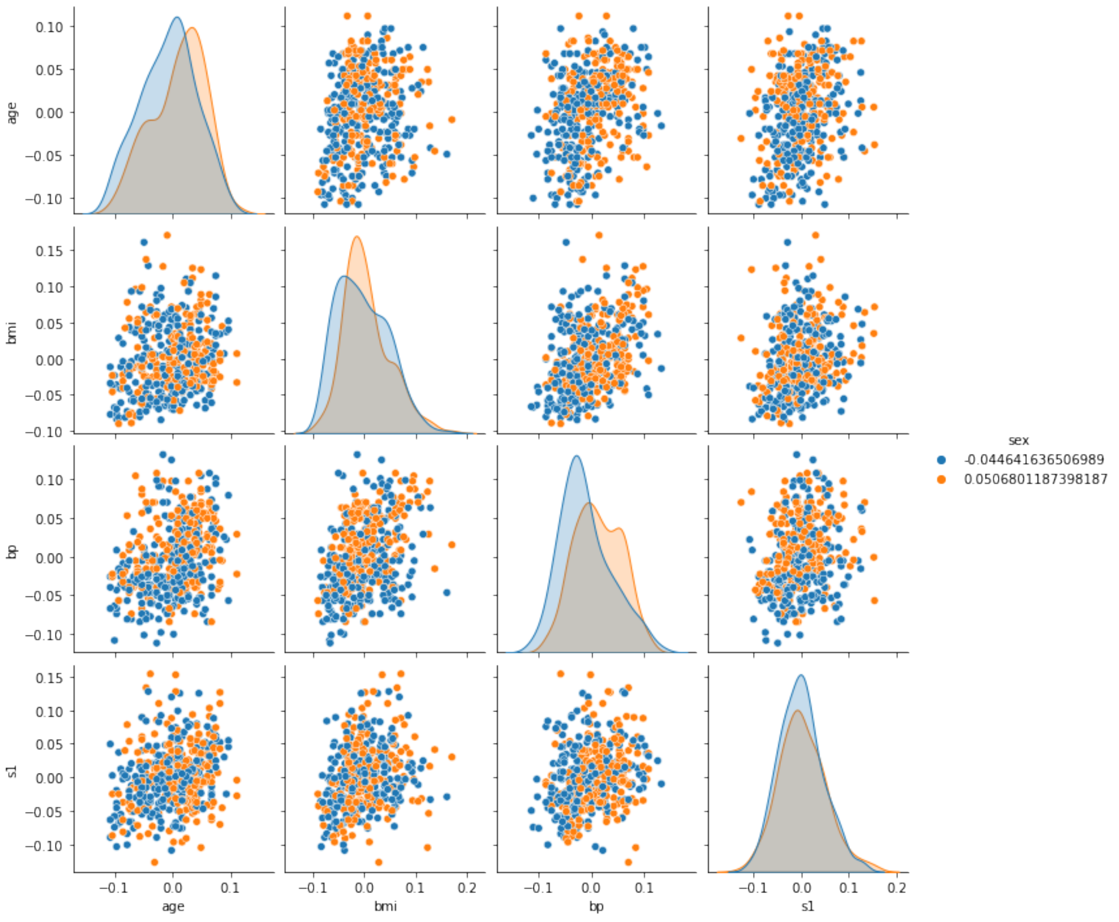
[4]:
```
(442, 11)
```

1.1 Notes on the dataset

- The data has been standerized (rescaled to have a mean of 0 and std of 1)
  - That's why there is -ve values in columns
- The categorical values in diabetes were recorded numerically as {0, 1} and then standardized
  - That's why the sex is incoded as numeric with -ve
- bmi is the body mass index, computed from height and weight, which is an approximation of body-fat percentage,
- bp is the blood pressure,
- s1–s6 are six blood serum measurements, and

- target is a numerical score measuring the progression of a patient's illness.

```
[13]: sns.pairplot(diabetes_df[['age', 'sex', 'bmi', 'bp', 's1']], hue='sex', palette␣
       ↪= "tab10");
```



2 Nearest-Neighbors Regression and Summary Statistics

We discussed nearest-neighbor classification in the previous chapter and we came up with the following sequence of steps: 1. Describe similarity between pairs of examples. 2. Pick several of the most-similar examples. 3. Combine the picked examples into a single answer.

As we shift our focus from predicting a class or category to predicting a numerical value, steps 1 and 2 can stay the same, However, when we get to step 3, we have to make adjustments nstead of simply voting for candidate answers,we need to combine the numerical values into a single, representative answer.

2.1 Measure of center: Mean & Median

- The median balances the count of values to the left and right

- The mean balances the total distances to the left and right

3 Building a k-nn Regressor model

[15]:
```
X_train, X_test, y_train, y_test =  skms.train_test_split(diabetes.data,␣
↪diabetes.target, test_size=.25)

knn = neighbors.KNeighborsRegressor(n_neighbors=3)
fit = knn.fit(X_train, y_train)
preds = fit.predict(X_test)

# evaluate our predictions against the held-back testing targets
metrics.mean_squared_error(y_test, preds)
```

[15]:
```
3108.805805805806
```

1. We built a different model: this time we used `KNeighborsRegressor` instead of `KNeighborsClassifier`.
2. We used a different evaluation metric: this time we used `mean_squared_error` instead of `accuracy_score`.

Both of these reflect the difference in the targets we are trying to predict—numerical values, not Boolean categories

[16]:
```
# target values range
diabetes_df['target'].max() - diabetes_df['target'].min()
```

[16]:
```
321.0
```

[41]:
```
# our prediction errors
np.sqrt(3500) # RMSE (root mean squared error)
```

[41]:
```
59.16079783099616
```

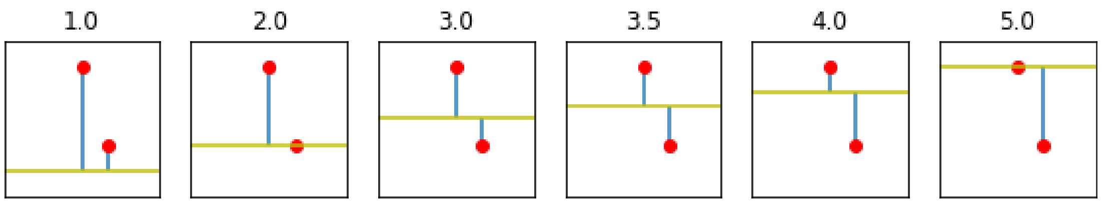
This means that our prediction was off by 60 unit, which is around 20% (60/320), or we have 80% accuracy

4 Linear Regression and Errors

4.1 No Flat Earth: Why We Need Slope

[19]:
```
def axis_helper(ax, lims):
    'clean up axes'
    ax.set_xlim(lims); ax.set_xticks([])
    ax.set_ylim(lims); ax.set_yticks([])
    ax.set_aspect('equal')
```

[24]:
```
# our data is very simple: two (x, y) points
D = np.array([[3, 5],
[4, 2]])

# we'll take x as our "input" and y as our "output"
x, y = D[:, 0], D[:, 1]

horizontal_lines = np.array([1, 2, 3, 3.5, 4, 5])

results = []
fig, axes = plt.subplots(1, 6, figsize=(10, 5))

for h_line, ax in zip(horizontal_lines, axes.flat):
    # styling
    axis_helper(ax, (0, 6))
    ax.set_title(str(h_line))

    # plot the data
    ax.plot(x, y, 'ro')  # The two red dots

    # plot the prediction line
    ax.axhline(h_line, color='y') # ax coords; defaults to 100%

    # plot the errors
    # the horizontal line *is* our prediction; renaming for clarity
    predictions = h_line
    ax.vlines(x, predictions, y)

    # calculate the error amounts and their sum of squares
    errors = y - predictions
    sse = np.dot(errors, errors)

    # put together some results in a tuple
    results.append((predictions, errors, errors.sum(), sse, np.sqrt(sse)))
```



[25]:
```
col_labels = "Prediction", "Errors", "Sum", "SSE", "Distance"
display(pd.DataFrame.from_records(results,
columns=col_labels,
index="Prediction"))
```

Prediction	Errors	Sum	SSE	Distance
1.0000	[4.0, 1.0]	5.0000	17.0000	4.1231
2.0000	[3.0, 0.0]	3.0000	9.0000	3.0000
3.0000	[2.0, -1.0]	1.0000	5.0000	2.2361
3.5000	[1.5, -1.5]	0.0000	4.5000	2.1213
4.0000	[1.0, -2.0]	-1.0000	5.0000	2.2361
5.0000	[0.0, -3.0]	-3.0000	9.0000	3.0000

We can see that in the prediction 3.5 the errors cancel itself and we don't want that so the best measure for this is the SSE (sum of squraed error)

What if we want the line to have a slope, instead of predicting the intercept now we also need to find the m (y = mx + b) - predicted = m*x + b - error = (m*x + b) - actual = predicted - actual - SSE = sum(errors^2) = sum(((m*x+b) - actual)^2 for x,actual in data) - total_distance = sqrt(SSE)

4.2 Performing Linear Regression

What if we have more than one feature, now we need to find the slope for each one. `we will name the slope a weight` So now the line will be y = $\sum_{x=1}^{n} w.x$

```
[26]: lr = linear_model.LinearRegression()
fit = lr.fit(X_train, y_train)
preds = fit.predict(X_test)

# evaluate our predictions against the unseen testing targets
metrics.mean_squared_error(y_test, preds)
```

```
[26]: 2428.04106442735
```

5 Optimization: Picking the Best Answer

Picking the best line means picking the best values for m and b or for the weights, Here are four strategies we can adopt: 1. **Random guess**: Try lots of possibilities at random, take the best one. 2. **Random step**: Try one line—pick an m and a b—at random, make several random adjustments, pick the adjustment that helps the most. Repeat. 3. **Smart step**: Try one line at random, see how it does, adjust it in some smart way. Repeat. 4. **Calculated shortcut**: Use fancy mathematics to prove that if Fact A, Fact B, and Fact C are all true, then the One Line To Rule Them All must be the best. Plug in some numbers and use The One Line To Rule Them All.

5.1 Random guess

```
[27]: # simple data to predict
tgt = np.array([3, 5, 8, 10, 12, 15])
```

[32]:
```
num_guesses = 10
results = []

for g in range(num_guesses):
    guess = np.random.uniform(low=tgt.min(), high=tgt.max()) # draw sample form␣
 ↪a unifom distribution over an interval
    total_dist = np.sum((tgt - guess)**2)  # SSE
    results.append((total_dist, guess))

best_guess = sorted(results)[0][1] # the smallest SSE (total dist, guess), and␣
 ↪get the guess
best_guess
```

[32]:
```
9.073247476312165
```

5.2 Random step

Method 2 starts with a single random guess, but then takes a random step up or down. If that step is an improvement, we keep it. Otherwise, we go back to where we were.

[38]:
```
# use a random choice to take a hypothetical
# step up or down: follow it, if it is an improvement
num_steps = 100
step_size = .05

best_guess = np.random.uniform(low=tgt.min(), high=tgt.max())
best_dist = np.sum((tgt - best_guess)**2)

for s in range(num_steps):
    new_guess = best_guess + (np.random.choice([+1, -1]) * step_size)
    new_dist = np.sum((tgt - new_guess)**2)
    if new_dist < best_dist:
        best_guess, best_dist = new_guess, new_dist
print(best_guess)
```

```
8.897353621662077
```

5.3 Smart step

[39]:
```
# hypothetically take both steps (up and down)
# choose the better of the two
# if it is an improvement, follow that step
num_steps = 1000
step_size = .02

best_guess = np.random.uniform(low=tgt.min(), high=tgt.max())
best_dist = np.sum((tgt - best_guess)**2)
```

```
print("start:", best_guess)

for s in range(num_steps):
    # np.newaxis is needed to align the minus
    guesses = best_guess + (np.array([-1, 1]) * step_size)
    dists = np.sum((tgt[:,np.newaxis] - guesses)**2, axis=0)

    better_idx = np.argmin(dists)

    if dists[better_idx] > best_dist:
        break

    best_guess = guesses[better_idx]
    best_dist = dists[better_idx]
print(" end:", best_guess)
```

```
start: 12.250936524771246
 end: 8.83093652477132
```

5.4 Calculated Shortcuts

To get the smallest SSE all we need is the mean

[42]:
```
print("mean:", np.mean(tgt))
```

```
mean: 8.833333333333334
```

6 Learning Performance

[49]:
```
# stand-alone code
from sklearn import (datasets, neighbors, model_selection as skms,␣
 ↪linear_model, metrics)

# Load the dataset
diabetes = datasets.load_diabetes()

# splitting to train and test
tts = skms.train_test_split(diabetes.data, diabetes.target, test_size=.25)
(diabetes_train, diabetes_test, diabetes_train_tgt, diabetes_test_tgt) = tts

# out two models
models = {'kNN': neighbors.KNeighborsRegressor(n_neighbors=3), 'linreg' :␣
 ↪linear_model.LinearRegression()}

for name, model in models.items():
    fit = model.fit(diabetes_train, diabetes_train_tgt)
    preds = fit.predict(diabetes_test)

    score = np.sqrt(metrics.mean_squared_error(diabetes_test_tgt, preds))
    print("{:>6s} : {:0.2f}".format(name,score))
```

```
   kNN : 67.33
linreg : 56.25
```