

Assignment 01 Companion Notebook

This notebook contains some examples that show off various properties of linear regression. This notebook is mostly focused on linear regression from a top-down perspective. That is, we are concerned with how does the algorithm behave rather than how it is implemented. As such we will be using a built-in solver for linear regression and treating it essentially as a black box (you will be opening that black box later in the assignment).

Numpy Practice

While you will not be doing any programming in this assignment, in order to understand the provided examples it helps to have at least a little bit of knowledge of `numpy`. Additionally, we will be using `numpy` extensively in this course. Please use the code block below to play around a bit with `numpy` (assuming you aren't already well-versed in `numpy`'s use').

In order to help you learn how to use `numpy`, you may consider consulting one of these tutorials.

- [Numpy for MATLAB Users](#)
- [Numpy Cheatsheet](#)

In [1]:

```
# play around with numpy here
import numpy as np
```

A Toy Linear Regression Problem

The notion of a toy problem is very useful for validating that a machine learning algorithm is working as it is intended to. The basic structure of a toy problem is as follows.

Suppose you are given a learning algorithm designed to estimate some model parameters \mathbf{w} from some training data (\mathbf{X}, \mathbf{y}) .

1. Generate values for the model parameters \mathbf{w} (e.g., set them to some known values or generate them randomly). If you were applying your algorithm to real data, you would of course not know these parameters, but instead estimate them from data. For our toy problem, we'll proceed with values that we generate so we can test our algorithms.
2. Generate some training input data, \mathbf{X} , (random numbers work well for this). Generate the training output data, \mathbf{y} , by applying the model with parameters \mathbf{w} . For example, for a linear regression problem if \mathbf{w} represents the regression coefficients, then we can generate each training label, y_i as $y_i = \mathbf{x}_i^T \mathbf{w}$.
3. Run your learning algorithms on the synthesized training data (\mathbf{X}, \mathbf{y}) to arrive at estimated values of the model parameters, $\hat{\mathbf{w}}$.
4. Compare \mathbf{w} and $\hat{\mathbf{w}}$ as a way of understanding whether your learning algorithm is working.

In the next code block, you'll see an example of a toy regression problem where we set $\mathbf{w} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$, generate some training data, and then recover \mathbf{w} by applying the linear regression algorithm.

In [2]:

```
import numpy as np

def linear_regression(X, y):
    w, _, _, _ = np.linalg.lstsq(X, y, rcond=-1)
    return w

n_points = 50
X = np.random.randn(n_points, 2)
w_true = np.array([1, 2])
y = X.dot(w_true)
w_estimated = linear_regression(X, y)

w_estimated
```

Out[2]:

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

Questions

1. What should be true about the relationship between `w_true` and `w_estimated` if the algorithm is working properly?
2. Are there values of `n_points` that would cause `w_true` to be different from `w_estimated`?

Investigating Noise

One thing you might be interested in knowing is how well the algorithm will work when the data doesn't perfectly conform to the hypothesized model (i.e., $y \neq \mathbf{x}^T \mathbf{w}$).

In the next code block we'll add some noise to the labels and see if the linear regression algorithm can still reconstruct the true parameters `w`. In particular, we going to add noise generated from a [Gaussian distribution](#) with a specified standard deviation. By modifying `noise_standard_deviation`, you can explore how the magnitude of noise influences the quality of the results (we'll be exploring standard deviation and Gaussians in more detail later in this course). By modifying `n_points` you can change how many training points are available to the linear regression model for parameter estimation.

In [3]:

```
n_points = 50
noise_standard_deviation = 2
X = np.random.randn(n_points, 2)
w_true = np.array([1, 2])
y = X.dot(w_true) + np.random.randn(n_points,) * noise_standard_deviation
w_estimated = linear_regression(X, y)

w_estimated
```

Out[3]:

```
array([1.08663159, 2.48604234])
```

Questions

1. Keeping `noise_standard_deviation` constant, characterize qualitatively the relationship between `n_points` and how close `w_estimated` is to `w_true`?
2. Keeping `n_points` constant, characterize qualitatively the relationship between `noise_standard_deviation` and how close `w_estimated` is to `w_true`?

Adding a y-intercept (bias) term

The beautiful thing about linear regression is that we can make it work for non-linear functions quite easily. The easiest way to do this is by augmenting the input data with additional features. In this way, functions that are non-linear in the original input space become linear in the augmented space.

For instance, the function $\mathbf{w}^T \mathbf{x} + b$ (where b is a scalar or bias term) is non-linear (We know it looks linear but it is actually affine. If

you are not convinced, you can test it against the properties of a [linear map](#)). On the other hand if we construct the vectors $\tilde{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}$

and $\tilde{\mathbf{w}} = \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix}$, then $\mathbf{w}^T \mathbf{x} + b = \tilde{\mathbf{w}}^T \tilde{\mathbf{x}}$ where $\tilde{\mathbf{w}}^T \tilde{\mathbf{x}}$ is now a linear function!

In the code block below, you'll see how this idea can be used to fit a linear regression model to model with a bias term.

In [4]:

```
b = 3
y = X.dot(w_true) + b
linear_regression(np.hstack((X, np.ones((X.shape[0], 1)))), y)
```

Out[4]:

```
array([1., 2., 3.])
```

Handling Other types of Non-linear Functions

Provided you know the form of the non-linear function you are fitting, you can use linear regression to fit arbitrary non-linear functions.

In the code below we will explore an example where the $y_i = w^T x_i + x_{i,1}^2$. We'll create modified feature vector $\tilde{\mathbf{x}} = \begin{bmatrix} x \\ x^2 \end{bmatrix}$, and apply linear regression to those vectors. You'll see that the linear regression with the additional feature will match much better to the training data (have a lower error) than the one where we don't allow for this new feature.

In [5]:

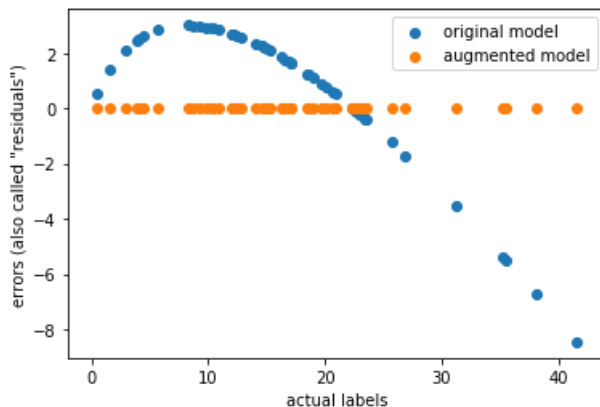
```
n_points = 50
X = np.random.randn(n_points,1) + 3
w_true = np.array([3])
y = X.dot(w_true) + np.square(X[:,0])
w_estimated = linear_regression(X, y)
original_model_preds = X.dot(w_estimated)
print("w_estimated", w_estimated)
print("Sum of squares for original model", np.sum(np.square(original_model_preds - y)))

X_augmented = np.column_stack((X, np.square(X[:,0])))
w_estimated_augmented = linear_regression(X_augmented, y)
print("w_estimated_augmented", w_estimated_augmented)

augmented_model_preds = X_augmented.dot(w_estimated_augmented)
print("Sum of squares for augmented model", np.sum(np.square(augmented_model_preds - y)))

%matplotlib inline
import matplotlib.pyplot as plt
plt.scatter(y, original_model_preds - y)
plt.scatter(y, augmented_model_preds - y)
plt.xlabel('actual labels')
plt.ylabel('errors (also called "residuals")')
plt.legend(['original model', 'augmented model'])
plt.show()
```

```
w_estimated [6.46433165]
Sum of squares for original model 367.300543489437
w_estimated_augmented [3. 1.]
Sum of squares for augmented model 2.1054327781799483e-27
```



Questions

1. Explain the observed differences in the sum of squares error for the two models. Why is the augmented model more accurate?
2. In the original model, the coefficient estimated for the input feature is much higher than the true value (roughly twice as large). Explain why this occurs and how adding the additional feature allows the augmented model to estimate the value accurately?
3. If you were applying linear regression to a real problem, how might you determine features to add to your model to capture non-linearities?

Predicting Bikeshare Data (Your First Machine Learning Application!)

The University of California maintains a repository of machine learning datasets. These datasets have been used over the years to benchmark algorithms in order to facilitate comparisons to previously published work. In this next section of the notebook you will be exploring the [Bikeshare dataset](#). The Bikeshare dataset contains daily usage data over a roughly two year period. Along with each

record of user counts, there are independent variables that measure various characteristics of the day in question (e.g., whether it was a weekday or a weekend, the air temperature, the wind speed).

Here is what the data looks like (Note: we're using Pandas for this example. We will be learning some Pandas later in the course, but for now don't worry about it).

In [6]:

```
import pandas as pd
bikeshare =
pd.read_csv('https://raw.githubusercontent.com/kylecho/nd101_pl_neural_network/master/Bike-Sharing-Dataset/day.csv')
bikeshare
```

Out[6]:

	instant	dteday	season	yr	mnth	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual
0	1	2011-01-01	1	0	1	0	6	0	2	0.344167	0.363625	0.805833	0.160446	331
1	2	2011-01-02	1	0	1	0	0	0	2	0.363478	0.353739	0.696087	0.248539	131
2	3	2011-01-03	1	0	1	0	1	1	1	0.196364	0.189405	0.437273	0.248309	120
3	4	2011-01-04	1	0	1	0	2	1	1	0.200000	0.212122	0.590435	0.160296	108
4	5	2011-01-05	1	0	1	0	3	1	1	0.226957	0.229270	0.436957	0.186900	82
5	6	2011-01-06	1	0	1	0	4	1	1	0.204348	0.233209	0.518261	0.089565	88
6	7	2011-01-07	1	0	1	0	5	1	2	0.196522	0.208839	0.498696	0.168726	148
7	8	2011-01-08	1	0	1	0	6	0	2	0.165000	0.162254	0.535833	0.266804	68
8	9	2011-01-09	1	0	1	0	0	0	1	0.138333	0.116175	0.434167	0.361950	54
9	10	2011-01-10	1	0	1	0	1	1	1	0.150833	0.150888	0.482917	0.223267	41
10	11	2011-01-11	1	0	1	0	2	1	2	0.169091	0.191464	0.686364	0.122132	43
11	12	2011-01-12	1	0	1	0	3	1	1	0.172727	0.160473	0.599545	0.304627	25
12	13	2011-01-13	1	0	1	0	4	1	1	0.165000	0.150883	0.470417	0.301000	38
13	14	2011-01-14	1	0	1	0	5	1	1	0.160870	0.188413	0.537826	0.126548	54
14	15	2011-01-15	1	0	1	0	6	0	2	0.233333	0.248112	0.498750	0.157963	222
15	16	2011-01-16	1	0	1	0	0	0	1	0.231667	0.234217	0.483750	0.188433	251
16	17	2011-01-17	1	0	1	1	1	0	2	0.175833	0.176771	0.537500	0.194017	117
17	18	2011-01-18	1	0	1	0	2	1	2	0.216667	0.232333	0.861667	0.146775	9
18	19	2011-01-19	1	0	1	0	3	1	2	0.292174	0.298422	0.741739	0.208317	78
19	20	2011-01-20	1	0	1	0	4	1	2	0.261667	0.255050	0.538333	0.195904	83
20	21	2011-01-21	1	0	1	0	5	1	1	0.177500	0.157833	0.457083	0.353242	75
21	22	2011-01-22	1	0	1	0	6	0	1	0.059130	0.079070	0.400000	0.171970	93
22	23	2011-01-23	1	0	1	0	0	0	1	0.096522	0.098839	0.436522	0.246600	150
23	24	2011-01-24	1	0	1	0	1	1	1	0.097391	0.117930	0.491739	0.158330	86
24	25	2011-01-25	1	0	1	0	2	1	2	0.223478	0.234526	0.616957	0.129796	186

		01-20												
instant	dteday	2011	season	yr	mnth	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual
25	26	2011-01-26	1	0	1	0	3	1	3	0.217500	0.203600	0.862500	0.293850	34
26	27	2011-01-27	1	0	1	0	4	1	1	0.195000	0.219700	0.687500	0.113837	15
27	28	2011-01-28	1	0	1	0	5	1	2	0.203478	0.223317	0.793043	0.123300	38
28	29	2011-01-29	1	0	1	0	6	0	1	0.196522	0.212126	0.651739	0.145365	123
29	30	2011-01-30	1	0	1	0	0	0	1	0.216522	0.250322	0.722174	0.073983	140
...
701	702	2012-12-02	4	1	12	0	0	0	2	0.347500	0.359208	0.823333	0.124379	892
702	703	2012-12-03	4	1	12	0	1	1	1	0.452500	0.455796	0.767500	0.082721	555
703	704	2012-12-04	4	1	12	0	2	1	1	0.475833	0.469054	0.733750	0.174129	551
704	705	2012-12-05	4	1	12	0	3	1	1	0.438333	0.428012	0.485000	0.324021	331
705	706	2012-12-06	4	1	12	0	4	1	1	0.255833	0.258204	0.508750	0.174754	340
706	707	2012-12-07	4	1	12	0	5	1	2	0.320833	0.321958	0.764167	0.130600	349
707	708	2012-12-08	4	1	12	0	6	0	2	0.381667	0.389508	0.911250	0.101379	1153
708	709	2012-12-09	4	1	12	0	0	0	2	0.384167	0.390146	0.905417	0.157975	441
709	710	2012-12-10	4	1	12	0	1	1	2	0.435833	0.435575	0.925000	0.190308	329
710	711	2012-12-11	4	1	12	0	2	1	2	0.353333	0.338363	0.596667	0.296037	282
711	712	2012-12-12	4	1	12	0	3	1	2	0.297500	0.297338	0.538333	0.162937	310
712	713	2012-12-13	4	1	12	0	4	1	1	0.295833	0.294188	0.485833	0.174129	425
713	714	2012-12-14	4	1	12	0	5	1	1	0.281667	0.294192	0.642917	0.131229	429
714	715	2012-12-15	4	1	12	0	6	0	1	0.324167	0.338383	0.650417	0.106350	767
715	716	2012-12-16	4	1	12	0	0	0	2	0.362500	0.369938	0.838750	0.100742	538
716	717	2012-12-17	4	1	12	0	1	1	2	0.393333	0.401500	0.907083	0.098258	212
717	718	2012-12-18	4	1	12	0	2	1	1	0.410833	0.409708	0.666250	0.221404	433
718	719	2012-12-19	4	1	12	0	3	1	1	0.332500	0.342162	0.625417	0.184092	333
719	720	2012-12-20	4	1	12	0	4	1	2	0.330000	0.335217	0.667917	0.132463	314
720	721	2012-12-21	1	1	12	0	5	1	2	0.326667	0.301767	0.556667	0.374383	221
721	722	2012-12-22	1	1	12	0	6	0	1	0.265833	0.236113	0.441250	0.407346	205
722	723	2012-12-23	1	1	12	0	0	0	1	0.245833	0.259471	0.515417	0.133083	408
723	724	2012-12-24	1	1	12	0	1	1	2	0.231304	0.258900	0.791304	0.077230	174
724	725	2012-12-25	1	1	12	1	2	0	2	0.291304	0.294465	0.734783	0.168726	440
725	726	2012-12-26	1	1	12	0	3	1	3	0.243333	0.220333	0.823333	0.316546	9
726	727	2012-12-27	1	1	12	0	4	1	2	0.254167	0.226642	0.652917	0.350133	247
727	728	2012-12-28	1	1	12	0	5	1	2	0.253333	0.255046	0.590000	0.155471	644

	instant	dteday	season	yr	mnth	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual
728	729	2012-12-29	1	1	12	0	6	0	2	0.253333	0.242400	0.752917	0.124383	159
729	730	2012-12-30	1	1	12	0	0	0	1	0.255833	0.231700	0.483333	0.350754	364
730	731	2012-12-31	1	1	12	0	1	1	2	0.215833	0.223487	0.577500	0.154846	439

731 rows × 16 columns

Next, we'll perform regression with the column `cnt` (number of daily riders) as the output variable and `season`, `yr`, `mnth`, `holiday`, `weekday`, `workingday`, `weathersit`, `temp`, `atemp`, `hum`, `windspeed`, and a constant (bias) term as independent (explanatory variables). Predicting the demand for the Bikeshare service on a given day could help to determine when to purchase new bikes.

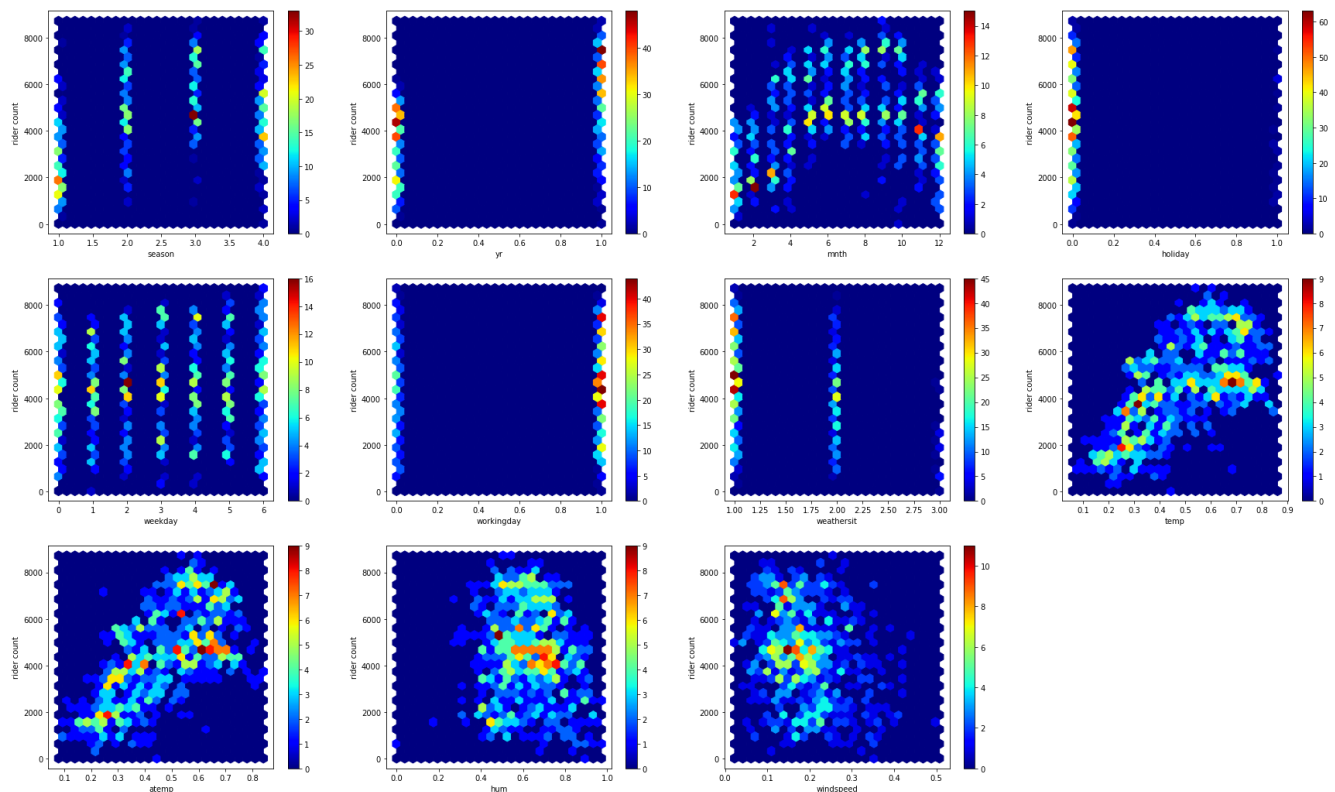
Before actually performing the regression, examine the hex plots below that show pairwise plots between each independent variable and the dependent variable (we avoided using scatter plots as those don't give a good sense of the density of the data when there are lots of overlaps).

In [7]:

```
X_bikeshare = bikeshare.drop(columns=['instant', 'dteday', 'cnt', 'registered', 'casual'])
y_bikeshare = bikeshare['cnt']

plt.figure(figsize=(30, 18))
for idx, col in enumerate(X_bikeshare):
    plt.subplot(3, 4, idx+1)
    plt.hexbin(X_bikeshare[col], y_bikeshare, gridsize=25, cmap='jet')
    plt.colorbar()
    plt.xlabel(col)
    plt.ylabel('rider count')

plt.subplots_adjust(wspace=.2)
plt.show()
```



Questions

- Before running the regression, make a prediction as to the sign of each of the coefficients for each of the explanatory variables. That is will the corresponding weight for each of these 11 variables be positive, negative, or close to 0?

Next, run the actual regression.

In [8]:

```
X_bikeshare['bias'] = 1  
list(zip(linear_regression(X_bikeshare, y_bikeshare), X_bikeshare.columns))
```

Out[8]:

```
[(509.775198288188, 'season'),  
(2040.703401658302, 'yr'),  
(-38.97956441091838, 'mnth'),  
(-518.9919312460555, 'holiday'),  
(69.06221629898982, 'weekday'),  
(120.35698921300694, 'workingday'),  
(-610.9870081089621, 'weathersit'),  
(2028.916103475132, 'temp'),  
(3573.2742884028607, 'atemp'),  
(-1018.8615712186565, 'hum'),  
(-2557.5691378686074, 'windspeed'),  
(1469.0030645879083, 'bias')]
```

Questions

1. Do the results match your predictions? If not, hazard a guess as to why they don't match (you won't necessarily be able to determine this given the information provided).

In [0]: