Assignment 1: Linear Regression

Machine Learning

Fall 2019

Learning Objectives

- Gain some familiarity with some of the key ideas in machine learning.
- Review of mathematical concepts we will be using in the beginning part of this course.
- Familiarize yourself with computational tools for machine learning.
- Learn linear regression using a "top-down" approach.

1 Six Big Ideas in Machine Learning

Before diving into the specifics of our first machine learning algorithm, let's examine some important ideas in machine learning. Why six big ideas? If Rob Martello has taught us anything, it's that all good things come in sixes.

Idea 1: Correlations for the Win?

ML algorithms learn to exploit correlations in data in order to make predictions. For instance, if one was using an ML algorithm to recognize whether someone was smiling in an image, the algorithm might learn that bright pixels around the mouth region are correlated with smiles (e.g., they indicate that a person's teeth are showing). This correlation would likely be useful for determining whether a new image of a face was a smile or not. Now suppose you take this model and apply it to a new dataset. You may find that faces that are angry are mistakenly marked as smiling! Why? In the case of angry facial expressions the teeth may also be showing. Of course you would expect the learning algorithm to be smart enough to realize that just using the presence of teeth is not enough to conclusively determine whether someone is smiling. Whether or not this actually happens is a function of the training data given to the ML algorithm. If, for instance, the training set was scraped from profile pictures from a dating website, the training set may not contain pictures of angry faces. Unfortunately, while exploiting correlations is one of the most powerful aspects of ML systems, it is also one of the most potentially problematic.

- *Example 1: Reinforcing Hiring Biases* You may have heard that Amazon scrapped a secret AI recruiting tool that showed bias against women. More specifically, the tool performed automatic keyword analysis of job applications to predict whether or not the applicant was worth forwarding on to a human for further evaluation. Early in the development of this system researchers discovered that the model the system had learned placed a negative weight on words such as "women's" as well as the names of some women's colleges. While there is of course no causal link between these words

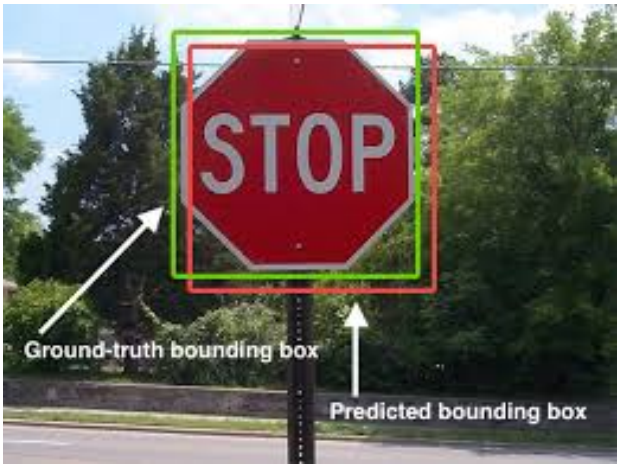
appearing in a job application and the suitability of the candidate for the job, there was a *correlation* in the training set between the presence of words such as "women's" and the candidates not being invited for interviews.

Why might such a correlation exist in the training set? There are many different possible explanations for this correlation, ranging from overt or unconscious bias in the applicant evaluators whose judgments helped form the training data to systemic discrimination that denies women equal access to educational opportunities in STEM. The important thing to take away from this is not *why* there was a correlation, but that the existence of the correlation in the training data caused the model to utilize the correlation in order to evaluate new data. Amazon realized that this was very bad and decided to take steps to address the problem (they say they never used the system to make actual job-screening decisions). Despite efforts to prevent the algorithm for exploiting such correlations, the group determined that they couldn't fully guarantee that the algorithm had not found another way to achieve the same discriminatory outcome and terminated the project.

- *Example 2: Adversarial Machine Learning*

  A second example of an ML algorithm exploiting correlations in training data in unexpected ways can be found in computer vision methods for object detection. Identifying salient visual objects such as road signs and pedestrians is an important building block for applications such as autonomous cars. A popular algorithm for this task, YOLO (You Only Look Once)[1], can identify and localize objects in images with surprising accuracy. For instance, in the image below YOLO identified the stop sign in the image successfully.

[1] Cool video of YOLO version 3, TED talk from the head YOLO researcher



While this all seems great, there is a catch. It is very difficult to understand *how* YOLO is making these predictions. That is, what is it about this image that causes the YOLO algorithm to be able to tell that it is a stop sign? Perhaps it is the white text on the red background. Perhaps it is the word "STOP." In fact, the network that makes this prediction is so complex, that it is impossible for us to say definitively exactly how it makes its decision. What we do know is that the model exploits

correlations in the training data between input features (pixels) and outputs (object locations) in potentially unpredictable ways.

The complexity of the model makes it vulnerable to bad actors (or adversaries). Researchers at University of Michigan used a form of ML known as *adversarial machine learning* to create a specially crafted sticker that could be attached to a stop sign that would make it invisible to the YOLO model (that is YOLO would not identify it as a stop sign). Clearly, this has major implications for the safety of using a model such as this in an application like a self-driving car. An example of the attack is shown in Figure 1.

Figure 1: A stop sign with a specially crafted sticker that causes a neural network to fail to identify it as a stop sign.

Idea 2: There's No Such Thing as a Free Lunch

> "All models are wrong, but some useful."
> — George Box

At the beginning of this document we have a reminder of the basic supervised machine learning setup. A one sentence statement of the setup is that we try to generalize from a set of training data to construct a function $\hat{f}^\star$ that best predicts the corresponding output data for unseen input data (e.g., predicting the facial expression of a face that was not in the training set based on a training set of sample faces). In the previous big idea, we discussed how machine learning could go wrong when there are correlations in the data that seem useful to the ML algorithm but are ultimately counterproductive to how we'd like the system to make decisions. It turns out that even before you choose the training data for your algorithm, you must provide an inductive bias to constrain the space of possible models you might fit. Examples of common inductive biases include the following (the previously linked article has some more).

1. The prediction function $\hat{f}^\star$ should change smoothly as you vary the input $\mathbf{x}$.
2. The prediction function has a particular form (e.g., linear).
3. The prediction function is sparse (it ignores the majority of the inputs).

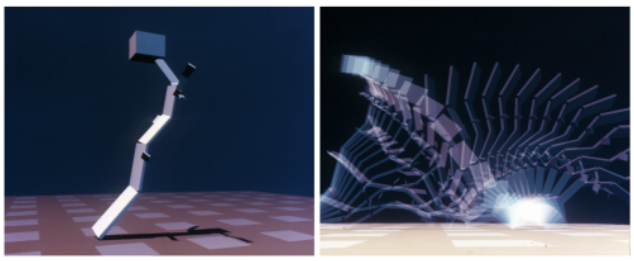
In fact, there are a whole class of theorems called No-Free-Lunch (NFL) theorems that state that without inductive biases (such as the ones stated above), learning from data is essentially impossible. This connects us back to the quote from George Box. While the inductive bias we encode into our model will never fully represent reality, having this bias is necessary to allow the model to do the useful work of making predictions. What's important for us as machine learning scientists and practitioners is to be explicit about the biases we are introducing when settling on a particular model so that we can best evaluate our results and predict the limitations of our systems.

Idea 3: It's All About How You Frame the Problem

Using Machine Learning algorithms can be a bit disorienting for someone used to the typical engineering workflow. A cartoon picture of the engineering workflow is that you are given a problem (perhaps it is initially difficult to solve or ambiguous), you might

reframe the problem to make it easier to solve, and then you work to devise a solution to the reframed problem. In machine learning, the last step is replaced by providing examples of how you'd like your system to work (i.e., input / output pairs), and then the creation of the actual system is automated by the ML algorithm! Your job as an ML practitioner is to reframe the original problem (both by specifying the form of the model and giving appropriate training data) so that the ML algorithm can compute a solution. If you've done the reframing properly, the solution to the reframed problem will also be a good solution to the original problem.

As an example of when a solution to the reframed problem would not be desirable, consider the use of a machine learning algorithm to teach a virtual character to walk in a simulated environment. You might reframe this problem for the ML algorithm as tasking it with computing a controller for the virtual character that moves the character's center of mass forward as fast as possible. The ML algorithm can now search over a vast space of possible control strategies to learn the one that most quickly propels the center of mass. However, it doesn't necessarily follow that this controller will result in the character walking using a normal bipedal gait.



The notion that the solution an algorithm finds might be unpredictable to the designer is known as "emergence." Some cool examples of this played out in actual experiments in evolving virtual creatures, which are summarized in the paper The Surprising Creativity of Digital Evolution. For instance, a virtual character learned that falling down, see picture above, and getting up was more efficient for locomotion than constantly hopping (which is what the designer had intended the system to learn).

For more examples of this sort of thing, consider checking out Karl Sims: Evolved Virtual Creatures or the short article When AI Surprises Us. This also connects back to the age-old debate over whether falling with style can be considered flying.

Idea 4: ML Systems Can Learn Intermediate Representations

In the next few weeks we'll learn about artificial neural networks (ANNs). ANNs are biologically inspired algorithms since their functioning, at an abstract level, is modeled on the functioning of biological neurons (e.g., in the brain).

ANNs accept input patterns at an array of virtual neurons called the input layer (see Figure 2). The neurons in the input layer are connected to other neurons via virtual axons that control to what extent a particular input neuron activates a downstream neuron. The second set of neurons, called the "hidden layer" (shown in blue in the middle of the figure), is responsible for computing intermediate, hidden representations of the input

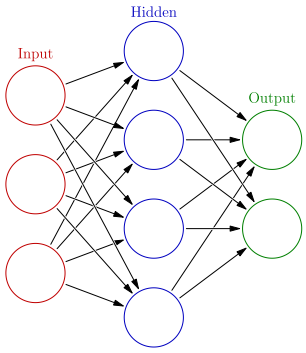


Figure 2: An artificial neural network with a single hidden layer.

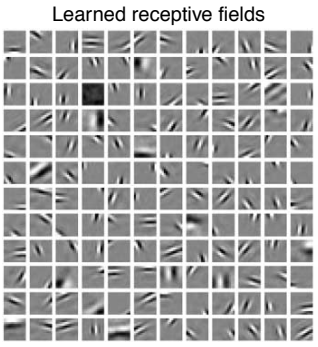
data. This process continues as activations propagate through the network until activations are generated at the output layer (shown in green on the right of the figure). These outputs could correspond to any salient properties of the input (e.g., if the input is an image, the output might encode the objects in the image).

What's amazing about ANNs is that there are learning algorithms for setting the connection strengths between these virtual neurons (the black arrows in Figure 2) based on training data (input / output pairs). These learning algorithms tune the connections strengths (also called "weights") such that for the provided training data the network produces the appropriate training outputs (e.g., if you show the network a training set of images of cats or dogs, over time the network will adjust its weights so that the output is "cat" when the network is presented an image of a cat and "dog" if presented an image of a dog). The algorithms used to tune the network weights are only concerned with reproducing the output patterns, the network is free to choose how it represents information within the network (i.e., at the hidden layer).

What's super amazing is that we can actually examine the internal representations of a neural network to understand how it's performing the computation from input to output. For instance, Figure 3[2] shows a visualization of the internal representations learned by a network trained to best compress a training set of images (these sorts of networks are called "auto-encoders"). The receptive fields of each of the hidden units in the network and can be understood as specifying how each input pixel activates a particular hidden unit (gray corresponds to no activation, black to negative activation, and white to positive activation). It's remarkable that these receptive fields have coherent structure: they are localized in space, tuned to particular orientations, and tuned to features at a particular scale. You can think of these as oriented edge detectors that the network learned completely on its own (it was never told to extract edges from the images in the training set).

What's super-duper amazing is that if we compare the receptive fields learned by the artificial neural network to the simple cells in the primary visual cortex of a cat, there are a number of striking similarities. Just as in the ANN, the biological neural network responds to edges at particular orientations and scales. The scientists Hubel and Wiesel performed the pioneering work in neuroscience to establish the properties of receptive fields in the primary visual cortex. Consider watching a video of their experiment that eventually garnered a Nobel prize (note that in the video the static sound corresponds to the measurement of spikes in activity of an individual neuron in the brain of an anesthetized cat).[3]. The implication of the similarity between the receptive fields of the neurons in the cat brain and the virtual neurons in the ANN is that they are similar because they are fundamentally solving the same problem (i.e., efficiently representing visual information). In this light, that they should find similar solutions to this problem is not as surprising as it may first seem.

Figure 3: 12x12 receptive fields learned from an neural network trained to optimally compress images



[2] From Sparse coding of sensory inputs

[3] There are a variety of opinions on the ethics of performing research on animals

Idea 5: Machine Learning Zoomed Out

Historically, most ML courses have been laser-focused on learning about learning algorithms (e.g., neural networks, support vector machines, decision trees, etc.). In some

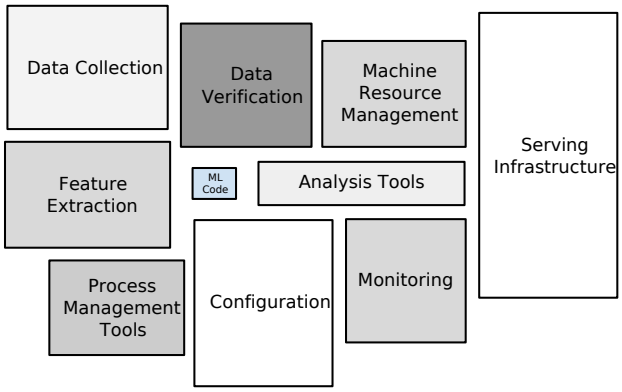
courses there would be a little bit of emphasis on machine learning applications, which have always been strongly tied to the research in ML algorithms and theory. The focus on ML algorithms also reflected the positioning of these courses within Computer Science curricula, which approached the field more from a liberal arts perspective rather than an engineering one.

A number of recent trends have made the almost sole focus on learning algorithms insufficient for those who want to either use ML in their careers or go into ML as a field.

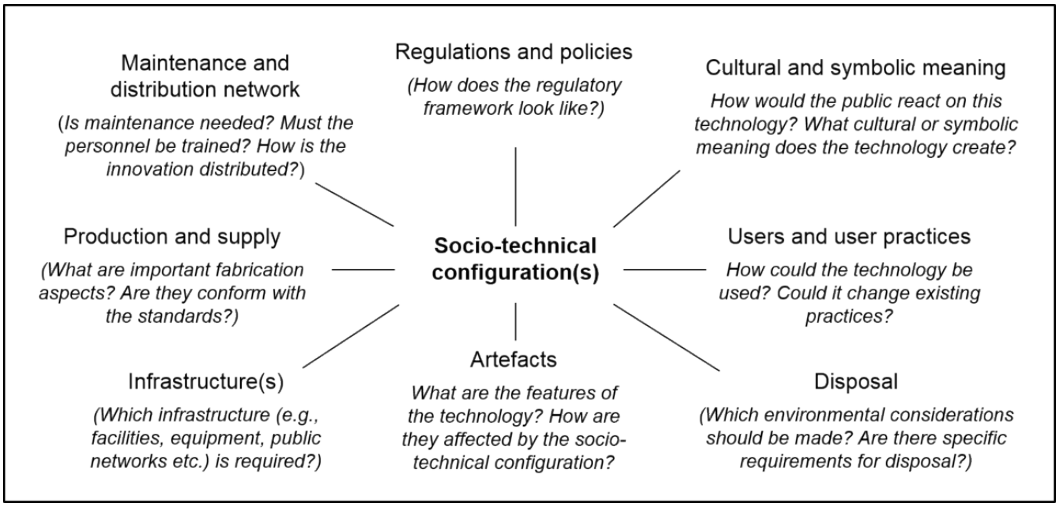
1. The explosion of data has made the skills necessary for collecting, wrangling, exploring, and cleaning data very relevant.

2. Improvements in the accuracy of ML algorithms coupled with the ability to deploy ML systems to a wide variety of devices (e.g., mobile phones) means that it is increasingly important to consider how ML systems will behave in real-world, highly complex settings.

The first point ties into a set of skills sometimes grouped under "Data Science." While we will have a comparatively lesser focus on this skillset than in our dedicated Data Science course, we will be learning some of these skills. The second point corresponds to ML systems as embedded in larger and more complex contexts. As you've seen from some of the examples earlier in this document, unexpected things can happen when ML algorithms meet messy and/or biased real world data (take for example the automated job applicant evaluator). In light of this, again, we think that the traditional focus on ML algorithms is not adequate for a modern class on ML. Here are two figures to further illustrate this point.



In the figure above, the box labeled *ML Code* is the actual learning algorithm. But in modern systems, this is but a small fraction of all of the tools needed to deploy a real world ML system. This is not to say that we will be spending a lot of time learning about each of these other boxes (we will learn about some of them), but it helps to have a sense of the software ecosystem in which your ML model would be deployed.



In addition to understanding how ML code is situated within larger software ecosystems, it is even more important to realize the socio-technical context in which an ML system is deployed. The figure above shows a socio-technical analysis of a technology. The figure highlights the need to consider contextual factors such as user impacts, culture, and regulations when analyzing technologies.

Using the tools of socio-technical systems analysis is becoming increasingly popular for analyzing machine learning systems. We'll be digging into some of these resources later in the course, but here are two papers in this spirit.

- Reframing AI Discourse
- Fairness and Abstraction in Sociotechnical Systems

Idea 6: It's Not All Doom and Gloom

While we'll be talking a lot about how ML can go wrong, unleashing unexpected consequences, we'll also be talking about the positive things that ML can do. Here are just a couple of resources that discuss such systems (not to say that these systems don't have the potential for things to go wrong!). We'll leave this list deliberately short to give you a chance to find your own example in the exercise below.

- AI for social good: 7 inspiring examples
- Machine Learning for Web Accessibility
- 19 Times Data Analysis Empowered Students and Schools
- This one is kind of cheating. Austin Vesiliza put together a list of links to AI for social good projects that you might use for inspiration.

The ⋖ icon means that you will be submitting your response to this exercise with your assignment. By default responses to exercises do not need to be submitted.

Exercise 1 (60 minutes)

Now, we want to hear from you!

(a) Choose one of the big ideas above and write a short response to it. Your response could incorporate something surprising you read, a thought-provoking question, your personal experience, an additional resource that builds upon or shifts the discussion. We hope that this reflection will help scaffold class discussions and get you thinking about your interests in the big space that is ML. Also, you have license from us to customize the structure of your response as you see fit. As a rough guide, you should aim for a response of a 1-2 paragraphs.

Solution

There's no one right answer here!

(b) Idea 6 talks about the idea of ML for positive impact. What is one example of an ML application (real or imagined) that you think would have the largest (or most unambiguously) positive impact on the world? Why? Alternatively, what is an example of an ML application (real or imagined) that no matter how carefully the designers approach it, should just not exist due to the harm it would cause the world? Why?

Solution

There's no one right answer here!

2 Mathematical Background

Notice

For the purposes of this class, we will be consistent with the notation we use. Of course, when we link to other resources, they may use other notation. If notation is different in a way that causes confusion, we will try to point out pitfalls you should watch out for. Please use this link to access our guide to our notation conventions.

External Resource(s) (60 minutes)

In order to engage with this assignment, you'll want to make sure you are familiar with the following concepts (links to resources embedded below):

- Vector-vector multiplication

  - Section 2.1 of Zico Kolter's Linear Algebra Review and Reference
- Matrix-vector multiplication
  - Section 2.2 of Zico Kolter's Linear Algebra Review and Reference
  - The first bits of the Khan academy video on Linear Transformations
- Partial derivatives and gradients
  - Khan Academy videos on partial derivatives: intro, graphical understanding, and formal definition
  - Khan Academy video on Gradient

Exercise 2 (20 minutes)

Here are some diagnostic problems to test some basic understanding of the concepts above.

(a) Suppose $f(x, y) = 2x \sin y + y^2 x^3$. Calculate $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$, and ∇f.

☆ Solution

$$\frac{\partial f}{\partial x} = 2 \sin y + 3y^2 x^2 \tag{1}$$

$$\frac{\partial f}{\partial y} = 2x \cos y + 2yx^3 \tag{2}$$

$$\nabla f = \begin{bmatrix} 2 \sin y + 3y^2 x^2 \\ 2x \cos y + 2yx^3 \end{bmatrix} \tag{3}$$

(b) Suppose $\mathbf{x} = \begin{bmatrix} 3 \\ -1 \\ 4 \end{bmatrix}$ and $\mathbf{y} = \begin{bmatrix} 2 \\ 7 \\ 4 \end{bmatrix}$. Calculate $\mathbf{x} \cdot \mathbf{y}$, $\mathbf{x}^\top \mathbf{y}$, and $\mathbf{x}\mathbf{y}^\top$.

☆ Solution

$$\mathbf{x} \cdot \mathbf{y} = 3 \times 2 + -1 \times 7 + 4 \times 4 = 15 \tag{4}$$

$$\mathbf{x}^\top \mathbf{y} = \mathbf{x} \cdot \mathbf{y} = 15 \tag{5}$$

$$\mathbf{x}\mathbf{y}^\top = \begin{bmatrix} 3 \times 2 & 3 \times 7 & 3 \times 4 \\ -1 \times 2 & -1 \times 7 & -1 \times 4 \\ 4 \times 2 & 4 \times 7 & 4 \times 4 \end{bmatrix} \tag{6}$$

$$= \begin{bmatrix} 6 & 21 & 12 \\ -2 & -7 & -4 \\ 8 & 28 & 16 \end{bmatrix} \tag{7}$$

(c) Let $\mathbf{A} = \begin{bmatrix} \mathbf{a_1} & \mathbf{a_2} & \dots & \mathbf{a_n} \end{bmatrix} = \begin{bmatrix} \mathbf{row_1}^\top \\ \mathbf{row_2}^\top \\ \vdots \\ \mathbf{row}_m^\top \end{bmatrix}$

(that is, the matrix $\mathbf{A}$ can either be thought of as consisting of the columns $\mathbf{a_1}, \dots, \mathbf{a_n}$ or the rows $\mathbf{row_1}^\top, \dots, \mathbf{row_m}^\top$). Let $\mathbf{v}$ be an arbitrary n-dimensional vector.

Compute $\mathbf{Av}$ in terms of $\mathbf{a_1}, \dots, \mathbf{a_n}$.

☆ Solution

$$\mathbf{Av} = \mathbf{v}_1\mathbf{a}_1 + \mathbf{v}_2\mathbf{a}_2 + \ldots + \mathbf{v}_n\mathbf{a}_n \tag{8}$$

Compute $\mathbf{Av}$ in terms of the rows of $\mathbf{row_1}, \dots, \mathbf{row_m}$.

☆ Solution

$$\mathbf{Av} = \begin{bmatrix} \mathbf{v} \cdot \mathbf{row}_1 \\ \mathbf{v} \cdot \mathbf{row}_2 \\ \vdots \\ \mathbf{v} \cdot \mathbf{row}_m \end{bmatrix} \tag{9}$$

3 Supervised Learning Problem Setup

Suppose you are given a training set of data points, $(\mathbf{x_1}, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)$ where each $\mathbf{x_i}$ represents an element of an input space (e.g., a d-dimensional feature vector) and each y_i represents an element of an output space (e.g., a scalar target value). In the supervised learning setting, your goal is to determine a function $\hat{f}$ that maps from the input space to the output space. For example, if we provide an input $\mathbf{x}$ to $\hat{f}$ it would generate the predicted output $\hat{y} = \hat{f}(\mathbf{x})$.

We typically also assume that there is some loss function, ℓ, that determines the amount of loss that a particular prediction $\hat{y}_i$ incurs due to a mismatch with the actual output y_i. We can define the best possible model, $\hat{f}^\star$ as the one that minimizes these losses over the training set. This notion can be expressed with the following equation (note: that arg min in the equation below just means the value that minimizes the expression inside of the arg min, e.g., $\arg\min_x (x-2)^2 = 2$, whereas $\min_x (x-2)^2 = 0$).

$$\hat{f}^\star = \arg\min_{\hat{f}} \sum_{i=1}^{n} \ell\left(\hat{f}(\mathbf{x_i}), y_i\right) \tag{10}$$

4 Linear Regression from the Top-Down

Motivation: Why Learn About Linear Regression?

Before we jump into the *what* of linear regression, let's spend a little bit of time talking about the *why* of linear regression. As you'll soon see, linear regression is among the simplest (perhaps *the* simplest) machine learning algorithm. It has many limitations, which you'll also see, but also a of ton strengths. **First, it is a great place to start when learning about machine learning** since the algorithm can be understood and implemented using a relatively small number of mathematical ideas (you'll be reviewing these ideas later in this assignment). In terms of the algorithm itself, it has the following very nice properties.

- **Transparent:** it's pretty easy to examine the model and understand how it arrives at its predictions.
- **Computational tractable:** models can be trained efficiently on datasets with large numbers of features and data points.
- **Easy to implement:** linear regression can be implemented using a number of different algorithms (e.g., gradient descent, closed-form solution). Even if the algorithm is not built into your favorite numerical computation library, the algorithm can be implemented in only a couple of lines of code.

For linear regression our input data, $\mathbf{x_i}$, are d-dimensional vectors (each entry of these vectors can be thought of as a feature), our output data, y_i, are scalars, and our prediction functions, $\hat{f}$, are all of the form $\hat{f}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} = \mathbf{w}^\top \mathbf{x} = \sum_{i=1}^{d} w_i x_i$ for some vector of weights $\mathbf{w}$ (you could think of $\hat{f}$ as also taking $\mathbf{w}$ as an input, e.g., writing $\hat{f}(\mathbf{x}, \mathbf{w})$. Most of the time we'll leave $\mathbf{w}$ as an implicit input: writing $\hat{f}(\mathbf{x})$).

In the function, $\hat{f}$, the elements of the vector $\mathbf{w}$ represent weights that multiply various dimensions of the input. For instance, if an element of $\mathbf{w}$ is high, that means that as the corresponding element of $\mathbf{x}$ increases, the prediction that $\hat{f}$ generates would also increase (you may want to mentally think through other cases, e.g., what would happen is the element of $\mathbf{x}$ decreases, or what would happen if the entry of $\mathbf{w}$ was large and negative). The products of the weights and the features are then summed to arrive at an overall prediction.

Given this model, we can now define our very first machine learning algorithm: ordinary least squares (OLS)! In the ordinary least squares algorithm, we use our training set to select the $\mathbf{w}$ that minimizes the sum of squared differences between the model's predictions and the training outputs. Thinking back to the supervised learning problem setup, this corresponds to choosing $\ell(y, \hat{y}) = (y - \hat{y})^2$. Therefore, the OLS algorithm will use the training data to select the optimal value of $\mathbf{w}$ (called $\mathbf{w}^\star$), which minimizes the sum of squared differences between the model's predictions and the training outputs.

$$\mathbf{w}^\star = \arg\min_{\mathbf{w}} \sum_{i=1}^{n} \ell\left(\hat{f}(\mathbf{x_i}, \mathbf{w}), y_i\right) \tag{11}$$

$$\mathbf{w}^\star = \arg\min_{\mathbf{w}} \sum_{i=1}^{n} \left(\hat{f}(\mathbf{x_i}, \mathbf{w}) - y_i\right)^2 \tag{12}$$

$$= \arg\min_{\mathbf{w}} \sum_{i=1}^{n} \left(\mathbf{w}^\top \mathbf{x_i} - y_i\right)^2 \tag{13}$$

⚠ Notice

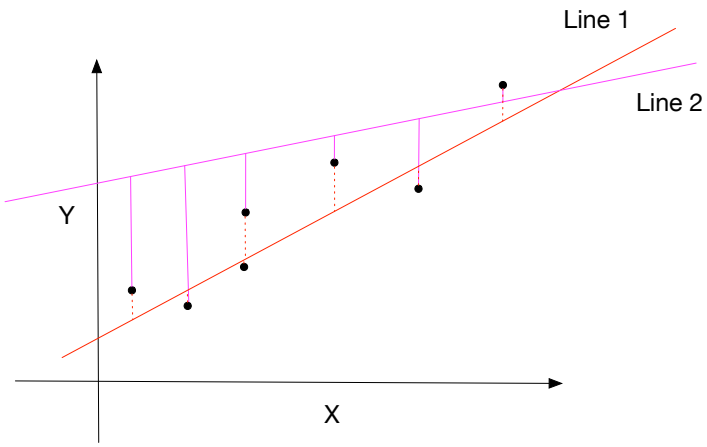
Digesting mathematical equations like this can be daunting, but your understanding will be increased by unpacking them carefully. Make sure you understand what was substituted and why in each of these lines. Make sure you understand what each symbol represents. If you are confused, ask for help (e.g., post on NB).

While we haven't talked at all about how to find $\mathbf{w}^\star$, that will be the focus of the next assignment, once we have $\mathbf{w}^\star$ we can predict a value for a new input point, $\mathbf{x}$, by predicting the corresponding (unknown) output, y, as $\hat{y} = \mathbf{w}^\star \cdot \mathbf{x}$. In this way, we have used the training data to learn how to make predictions about unseen data, which is the hallmark of supervised machine learning!

Exercise 3

Draw a scatter plot in 2D (the x-axis is the independent variable and the y-axis is the dependent variable). Fill in a few data points and draw a potential line of best fit. On the plot identify the vertical differences between the data points and the line (these differences are the residuals). Draw a second potential line of best fit and mark the residuals. From the point of view of ordinary least-squares, which of these lines is better?

☆ Solution



The red line (line 1) would be better since the residuals are generally smaller. Line 2 also has several large residuals, which when squared will cause a large penalty for line 2.

5 Getting a Feel for Linear Regression

In this class we'll be learning about algorithms using both a top-down and a bottom-up approach. By bottom-up we mean applying various mathematical rules to derive a solution to a problem and only then trying to understand how to apply it and how it well it might work for various problems. By top-down we mean starting by applying the algorithm to various problems and through these applications gaining a sense of the algorithm's properties. We'll start our investigation of linear regression using a **top-down approach**.

5.1 Linear Regression with One Input Variable: Line of Best Fit

If any of what we've said so far sounds familiar, it is likely because you have seen the idea of a line of best fit in some previous class. To understand more intuitively what the OLS algorithm is doing, we want you to investigate its behavior when there is a single input variable (i.e., you are computing a line of best fit).

External Resource(s)

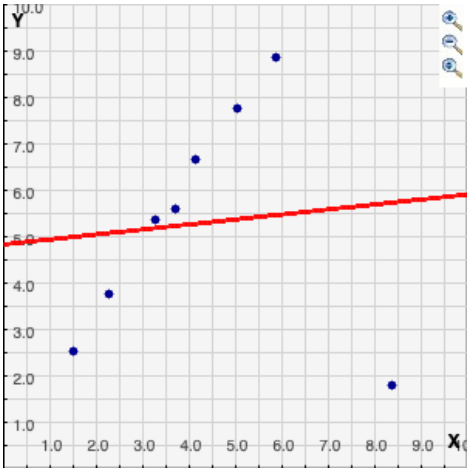
Use the line of best fit online app to create some datasets, guess the line of best fit, and then compare the results to the OLS solution (line of best fit).

Exercise 4

(a) Examine the role that outliers play in determining the line of best fit. Does OLS seem sensitive or insensitive to the presence of outliers in the data?

☆ Solution

OLS is very sensitive to outliers. A single outlier can change the slope of the line of best fit dramatically. Here is an example of this phenomenon.



(b) Were there any times when the line of best fit didn't seem to really be "best" (e.g., it didn't seem to capture the trends in the data)?

☆ Solution

This could happen for many reasons. If the dataset is pieceweise linear (e.g., composed of multiple line segments), if it has some other non-linear form (e.g., if it is quadratic), or if there are outliers.

5.2 Linear Regression with Multiple Input Variables: Explorations in Python

External Resource(s) (90 minutes)

Work through the Assignment 1 Companion Notebook to get some practice with `numpy` and explore linear regression using a top-down approach. You can place your answers directly in the Jupyter notebook so that you have them for your records.

Open in Colab

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 $y_i = \mathbf{x_i}^\top \mathbf{w}$.
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]:

Out[2]:

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

Notebook Exercise 1

(a) What should be true about the relationship between `w_true` and `w_estimated` if the algorithm is working properly?

(b) Are there values of `n_points` that would cause `w_true` to be different from `w_estimated'? (this is a bit tricky, so try some corner cases, values at the extremes, and see if you get a different result. If you do, try ot understand why.)

Solution

(a) These two vectors should be equal since the inferences of the model should match the true parameters.

(b) `n_points = 1` will give you a value for `w_estimated` that doesn't match `w_true`. This is because them problem is underconstrained (there are more unknowns than equations). Thus, the linear regression method can find multiple solutions that drive the cost to 0 and has no way of knowing which the correct one was.

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., $y \neq \mathbf{x}^\top \mathbf{w}$).

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 $\mathbf{w}$. In particular, we are 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 the noise influences the quality of the results (we'll be exploring standard deviation and Gaussians in more detail later in this course). For now you can just think of `noise_standard_deviation` as controlling the magnitude of corruption applied to the training outputs.

Additionally, 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([1.21351413, 2.18360576])
```

Notebook Exercise 2

(a) Keeping `noise_standard_deviation` constant, characterize qualitatively the relationship between `n_points` and how close `w_estimated` is to `w_true`? Why do you think this is the case (you don't have the formal language to describe this yet, so try to give a conceptual argument)?

(b) Keeping `n_points` constant, characterize qualitatively the relationship between `noise_standard_deviation` and how close `w_estimated` is to `w_true`? Why do you think this is the case (you don't have the formal language to describe this yet, so try to give a conceptual argument)?

Solution

(a) as `n_points` gets higher `w_estimated` gets closer to `w_true`. This makes sense intuitively since the random noise added to the training outputs `y` gets averaged out as more training poitns are provided.

(b) as `noise_standard_deviation` gets higher `w_estimated` gets farther from `w_true`. This makes sense intuitively since the random noise added to each to training output is greater and thus has the parameter estimates from linear regression are noisier as well.

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 agumented space.

For instance, the function $\mathbf{w}^\top \mathbf{x} + b$ (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 $\mathbf{w}^\top \mathbf{x} + b = \tilde{\mathbf{w}}^\top \tilde{\mathbf{x}}$ where $\tilde{\mathbf{w}}^\top \tilde{\mathbf{x}}$ 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 $y_i = w^\top x_i + x_{i,1}^2$. 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.

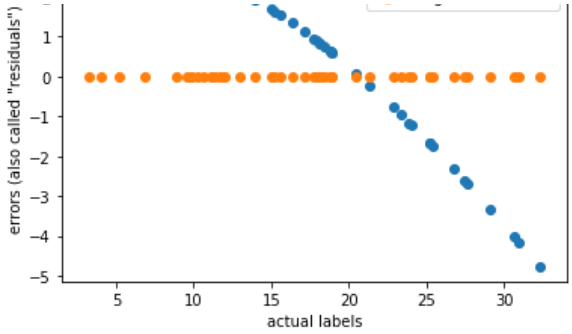
In [5]:

```
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 [6.28834633]
Sum of squares for original model 233.8487631525443
w_estimated_augmented [3. 1.]
Sum of squares for augmented model 4.599059077438499e-28
```



Notebook Exercise 3

(a) Explain the observed differences in the sum of squares error for the two models. Why is the augmented model more accurate?

(b) 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 why the augmented model doesn't have this problem?

(c) If you were applying linear regression to a real problem, how might you determine features to add to your model to capture non-linearities? There are many right answers to this, so you should be thinking of this as an open-ended question rather than one with a specific right answer.

Solution

(a) The augmented model is much more accurate (has a lower sum of squares) because due to the additional feature it can model the non-linear behavior in the training data.

(b) The original model is forced to artificially inflate the weight placed on the input feature to compensate for not being able to model the non-linear (quadratic behavior). It winds up overestimating the outputs for low input values and underestimating them for high input values. Since the augmented model has the quadratic feature, it can properly assign weight to that feature and keep the weight on the original feature at the appropriate value.

(c) There are lots of ways to do this. Here are three (we know you'll come up with more good ideas):

- See if there are any patterns in the plot of model residuals (e.g., the plot above shows a quadratic pattern in the errors, which might point towards the need to add a quadratic feature).
- Use domain-specific knowledge. For instance, there might be a reason why you expect a particular non-linear behavior to show up (e.g., you are modeling data that is periodic, such as temperature fluctuations over the year, and you know there is some sinusoidal behavior).
- You can use brute force search to try a bunch of different non-linearities and see which ones fit the data the best (there are a lot of machine learning models that work like this).

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 algoirthms in order to faciltate 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'll 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_p1_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

	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
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

	instant	dteday	season	yr	mnth	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual
703	704	2012-12-04	4	1	12	0	2	1	1	0.475833	0.468901	0.737500	0.174129	554
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
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). Why might we do this you ask? One potential reason would be to predict when to purchase or move bikes to meet anticipated demand.

Before actually performing the regression, examine the hex plots below that show pairwise plots between each independent

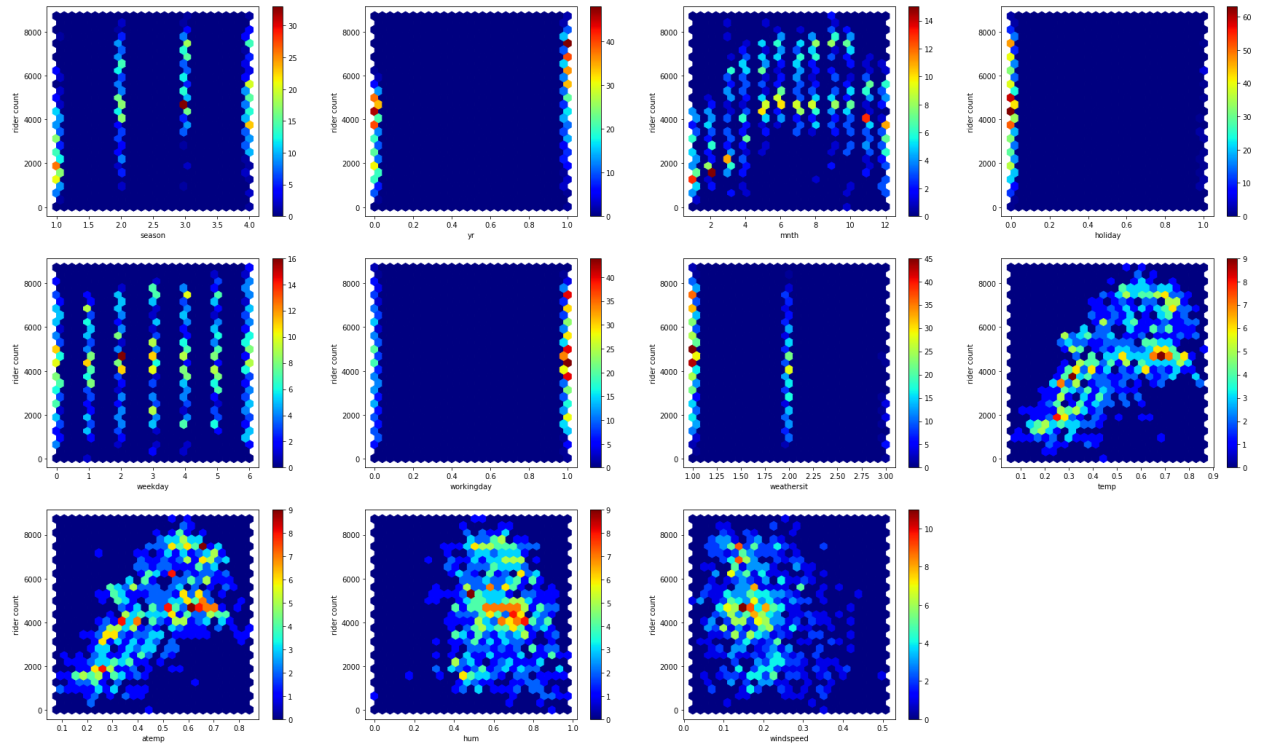
variable and the dependent variable (we avoided using scatter plots as those don't give a good sense of the density of the data when there are lots of overlapping points).

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()
```



Notebook Exercise 4

(a) 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?

Solution

(a) Based on what we estimate to be the linear trend in each scatter plot, we make the following predictions.

- season: positive
- yr: positive
- month: weakly positive (there is some downward movement at the end)
- weekday: close to 0
- workingday: positive
- holiday: negative
- weathersit: negative
- temp: positive
- atemp: positive
- hum: close to 0
- windspeed: weakly negative

Notebook Exercise 4 (part 2)

Next, run the actual regression.

(b) 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 [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')]
```

Solution

(b) They mostly match. Here are some more observations.

- Windspeed appeared to have a bigger influence (in the negative direction) than we intitially thought.
- `holiday` was hard to estimate mostly due to the visualization of the data as a hexplot (there isn't a lot of data in the `holiday = 1` part of the graph and thus it gets washed out in the plot. Using a different plot like a box plot could be better for this.
- `mnth` was a surprise. This looks like a few outliers at the end of the graph (`mnth` = 12) caused the weight to shift down and become negative.

It's also important to keep in mind that just because a variable is positively correlated with the output, does not imply that the OLS weight is positive. There are situations that arise when the independent variables are correlated with each other when this would not be the case. We'll be diving more into this in the next assignment.