
Pokemon
A (basic) introduction to classifiers

Introduction

There are like a million tutorials on ML, and that's awesome. I like Aurelien Geron's book (Hands on Machine Learning) best, but the courses Andrew Ng (there's one on Coursera) gives are great. So is fast.ai (that's the name of the wbsite) for deep learning. Chris Bishop's book (Pattern Recognition...) is what you should go for if you're interested in theory. Obviously if you would like to actually learn about ML theory, then those resources are the place to go, not to some random student's code. This is just a very simple taster problem that introduces some key concepts. I wrote the tutorial because when actually doing a new ML project, most coding will concern dataset building. That includes web scraping and data sanitization. It turns out most of the code for this is super ugly, and tends to be hidden away. But since it's hidden away, it's not obvious that you can actually acquire new datasets, sanitize them, then throw ML at the problem you care about to get insights. (Also everyone uses the same dataset -- MNIST, CFAR, etc, so it gets boring fast). I chose a dataset of Pokemon pixel art for several reasons: 1. It's fun 1. There aren't many pokemon (like 700 or so). Since the dataset is small and varied, simple classical ML models with feature engineering often perform better than systems which use more complicated models. We are scientists with a finite amount of microscopy / AFM / SAXS data, even if we datamine the literature. So having 700 varied samples is representative. We'll also have to introduce ideas like regularization to avoid overfitting. 1. The distribution of Pokemon types is highly unbalanced. There aren't many ghost types, and there are loads of normal types. This means that we have to introduce ideas like stratified test-train split etc. to handle the unbalanced data. It also means that evaluation of loss is a bit subtle.

The toy problem

Pokemon is a fictional TV / movie / game / merchandise francise. Maybe you've heard of it. Within the universe, sentient magical creatures are enslaved and take part in an unregulared version of the UFC. Also the slave owners are like 10 years old or something.

Each Pokemon is has an attribute, called "type", which is a categorical variable, being one of the set ['water','fire','normal','grass','bug',...]. Some Pokemon also have hybrid types, but I'm going to just ignore those. Each Pokemon also has a visual depiction, consisting of a bitmap (i.e. a 40x40x3 array of ints).

My challenge is: **Teach a machine to predict the Pokemon type from image** .

Prerequisities

I have written the code like I would do for an actual real life problem, so I use quite a few features of Python without trying to dumb down. The MIT course 6.0001 covers lots, but not all of the Python I'll use... in addition to those basics, you'll need to undersand OOP in Python, the map / reduce / filter system and lambda functions. Hopefully everything's obvious in context though...

Concepts this tutorial will cover

1. The different kinds of ML (or really statistical inference) problem
2. Test train split (how do you know if your classifier is working? what is generalization?)
3. What is a classifier?
4. Loss functions (what is a loss function? Are there reasons not to use naive accuracy?)
5. In sample vs out of sample error
6. Simple feature engineering

Plotting defaults and imports...

In [1]:
```
from matplotlib import rc
import matplotlib

font = {'family' : 'sans-serif',
        'weight' : 'normal',
        'size'   : 12}

matplotlib.rc('font', **font)
```

In [2]:
```
import requests
import os
from matplotlib import pyplot as plt
import numpy as np
import imageio
import re
```

Pokemon class: contains all relevant information: name, element (fire, water etc). Also contains image src, so can do automate downloads etc.

The next block creates a custom class for Pokemon. It then downloads some data from the website Bulbapedia, including a list of all Pokemon, and their bitmaps, https://bulbapedia.bulbagarden.net/wiki/Main_Page.

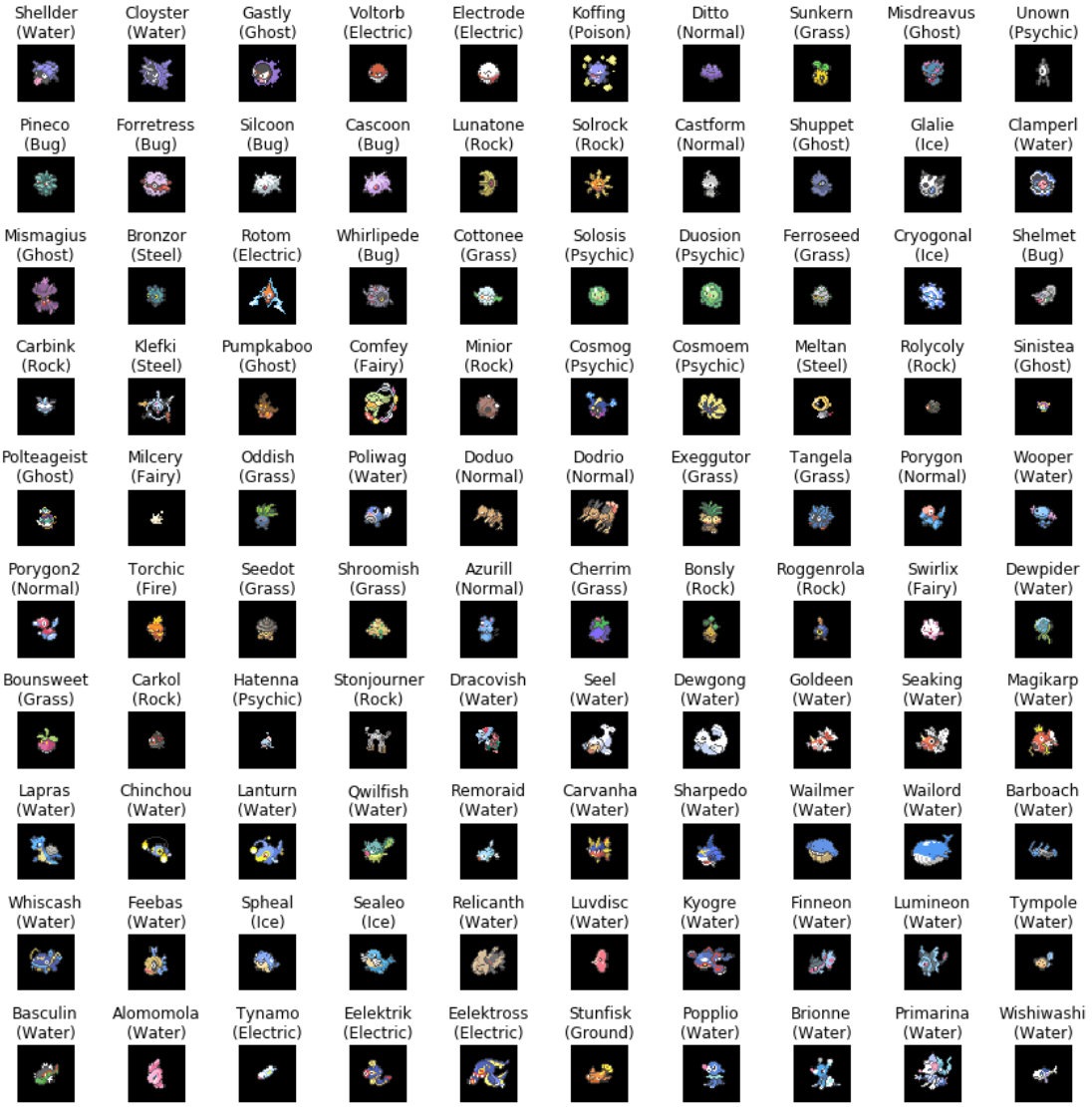
In [3]:

```
class Pokemon():
    '''Contains information on each Pokemon: need name, image src
(string), and element to initalize.
    Methods allow loading of image (i.e. downloading), conversion
of data to an array, and appropriate masking.'''
    def __init__(self,name,image_src,element):
        self.name = name
        self.image_src = image_src
        self.element = element
    def load_image(self):
        if not os.path.isfile("data/"+self.name+'.png'):
            with open("data/"+self.name+'.png','wb+') as f:
                data = requests.get('http://'+self.image_src).con
tent
                f.write(data)
    def get_array(self):
        full_image = imageio.imread("data/"+self.name+".png",pilm
ode = "RGBA")
        mask = full_image[:,:,-1] == 255
        for i in range(3):
            full_image[:,:,i] = full_image[:,:,i] * mask
        self.array = full_image[:,:,:3]

def get_pokemon_list():

    url = "https://bulbapedia.bulbagarden.net/wiki/List_of_Pok%C
3%A9mon_by_body_style"
    response = requests.get(url)

    with open('notes','w+') as f:
        f.write(response.text)

    dat = response.text
    x = re.findall('span class="plainlinks"><a href="/wiki/.*</a
>.*\n.*\n.*</a>.*\n.*\n.*\n',dat)
    list_of_pokemon = []

    for i in range(0,len(x)):

        src = re.findall('src=".*.png"',x[i])[0][7:-1]
        name_ = re.findall(r'title=".*mon',x[i])[0][7:-9]
        element = re.findall(r'title=".*type',x[i])[0][7:-6]

        pokemon = Pokemon(name_,src,element)
        list_of_pokemon.append(pokemon)

    for index,item in enumerate(list_of_pokemon):
        item.load_image()
        if not index % 100 : print (f"loaded {index} pokemon")
        item.get_array()

    return list_of_pokemon

list_of_pokemon = get_pokemon_list()
```

```
loaded 0 pokemon
loaded 100 pokemon
loaded 200 pokemon
loaded 300 pokemon
loaded 400 pokemon
loaded 500 pokemon
loaded 600 pokemon
loaded 700 pokemon
loaded 800 pokemon
loaded 900 pokemon
```

Let's have a look at a random selection of 100 Pokemon to get an idea of what our images are like...

In [4]:
```
f,ax = plt.subplots(10,10,figsize = (12,12) )

for index,ax_ in enumerate(ax.flatten()):
    ax_.margins(x=0)
    ax_.imshow(list_of_pokemon[index].array)
    ax_.set_title(f"{list_of_pokemon[index].name}\n({list_of_poke
mon[index].element})" )
    ax_.axis('off')

plt.subplots_adjust(wspace = 0.0)
plt.tight_layout()
```



An introduction to the different kinds of ML problem

Our objective is to predict the type of a Pokemon given its sprite. To start with, we'll focus on just four different types: fire, water, normal, grass.

What kind of ML problem is this?

Different algorithms are suited to different categories of ML problems. (TBH, these probably should be called something like "systems", because algorithm has a specific meaning in computer science.) So the first step in trying to solve a problem in ML is identifying what category of problem it is.

The objective of some ML problems is *regression* (outputting a continuous output, like 0.1, or 0.4 or 100). The objective of other ML problems *classification* (outputting a categorical output, like "cat", "dog", "email spam", "not email spam", possibly along with their probabilities). This problem is about classification.

Some ML problems have pre-labelled datasets. This would be perhaps 100 images, all with captions saying things like "cat" or "dog". Or 100 emails, where each one is labelled with "spam" or "not spam" or "maybe spam". These are called supervised problems. Alternatively, we may be given a thousand images of bugs, and asked to group them into clusters according to species, *despite* having no labels -- we have no idea which bug is which species. Instead, maybe of all of the images of bugs, some are very similar to each other and some are dissimilar. The job of the prediction system is then to partition the dataset into clusters. These are called *unsupervised* problems.

So our we need to develop a *supervised classification* model for our data.

To do this, I'll mostly use libraries from scikit-learn -- https://scikit-learn.org/stable/ (https://scikit-learn.org/stable/) (abbreviated sklearn). If you click that link there are loads of additional tutorials on regression / classification / clustering etc.

(N.B: there are also *generative* problems -- how do you make an algorithm generate photos of new faces if it's seen loads before.)

This tutorial is only going to try to classify Fire, Water, Grass, and Normal type Pokemon, and will only operate on the "small" bitmaps (which typically appear for non-legendary pokemon)

In [5]:
```
valid_pokemon = list(
    filter(lambda x : len(x.array) == 40, #filter by size
    filter(lambda x : (x.element in ['Fire','Water','Grass','Normal']),
                            list_of_pokemon))) # filter by type

X = np.array([x.array for x in valid_pokemon])
y = np.array([x.element for x in valid_pokemon])
```

Elements of X, which physically correspond to our images, consist of arrays of dimension 40x40x3. Elements of y correspond to strings, being "Fire", "Water" etc. We typically use the capital letter for X because its elements are vectors, (of matrices), so its tensor rank is at least 2. We typically use the lower case for y because it's elements are 0 dimensional.

Let's now have a look at what examples from each different class looks like:

In [6]:
```
f,ax = plt.subplots(8,10,figsize = (12,12) )

split_axis = np.split(ax.flatten(),4)

for pokemon_type,subaxis in zip(['Fire','Water','Grass','Normal
'],split_axis):

    for pokemon,ax_ in zip(filter(lambda x : x.element == pokemon
_type, list_of_pokemon),
                           subaxis):

        ax_.margins(x=0)
        ax_.imshow(pokemon.array)

        ax_.set_title(f"{pokemon.name}\n({pokemon.element})" )
        ax_.axis('off')

plt.subplots_adjust(wspace = 0.0)
plt.tight_layout()
```

Workflow for optimization

If our classification system is working, it should be able to predict the types of Pokemon it hasn't seen before. We also use the verb *generalize* to indicate that a system has learned a set of rules to map between inputs and outputs that remain valid on data it hasn't seen before. So how to we know how the classifier will perform on data it hasn't seen? To do this, we first split the data into a training set (which we feed to the ML classifier), and a test set (which we forbid the classifier from look at until we want to evaluate performance). So the Machine Learning workflow is the following: 0. Preprocess the input data (features): this may include rescaling, shifting, or changing the structure of matrices. 1. Split the dataset of Pokemon into a training dataset (67% of data), and a test dataset (33 % of data). We then hide the test dataset away. We don't have to choose that ratio of test to train data, but it's a good start. 1. Propose some model, $\mathcal{M}$, which is a set of maps between the image data and the categorical output ('Fire', 'Water' etc.) I'm saying a *set* of maps, because the precise map (i.e. which images correspond to which categorical type), is governed by a set of model parameters, $\vec{\theta}$. For a simple example: if we wanted to do linear regression, then the model would be the set of all possible straight lines. One particular straight line is specified by the value of the intercept and gradient; this is what I mean by the set of model parameters, which I'm carrying around in the vector $\vec{\theta}$. 1. Define a loss function, $L(\vec{\theta}, y_{pred}(\vec{\theta}), y)$. This takes the parameters, $\vec{\theta}$, the predicted output, y_{pred}, and the true (ground truth) output data, y. The loss function is a scalar which corresponds to how much the model sucks. You have to choose this, along with the model, and you have to use your intuition about what consitutes a good loss function. For linear regression, you might choose to minimise the sum of squared error. For a binary classification problem, you might want to minimize the number of misclassified examples. 1. Do numerical optimization to minimize $L(\vec{\theta}, y_{pred}, y)$, by varying θ. The notation for this is $\vec{\theta}^{(opt.)} = \text{argmin}_{\vec{\theta}} \left[L(\vec{\theta}, y_{pred}(\vec{\theta}), y) \right]$. We only use the training dataset to do this minimization! 1. Evaluate $L(\vec{\theta}^{(opt)}, y_{pred}(\vec{\theta}^{(opt)}), y)$, where y is the in sample error. Now, we see if the model has learned something! If the model has learned something, then when we apply the model which has been trained on the training dataset to the unseen test dataset, then we'll get a decent value of the loss function. If it does, the model is said to generalize, and we can be confident that when the model (with this set of parameters) is fed input it's never seen before, we'll get a reasonable value for the output. So basically what we're doing is: 1. Test train split 1. Propose a model (like linear regression, or logistic regression, or a neural network, or whatever) that has some parameters 1. Choose a loss function (like least squares, or cross entropy or whatever) that reflects your intuition about what good predictions are. 1. Change the parameters of the model until the loss function is minimized while optimizing only on the training dataset 1. See if the model which has been trained on the training dataset performs okay on the test dataset.

In [7]:
```
type_to_int = {'Normal':0,
               'Fire':1,
               'Grass':2,
               'Water':3}
int_to_type = {type_to_int[val] : val for val in type_to_int}
```

In [8]:
```
from sklearn.model_selection import train_test_split

def preprocess_data(X,y):

    X_flat = list([x.flatten()/255. for x in X]) #N.B: I'm flatte
ning all of the data...
    y_2 = list(map(lambda x : {'Normal':0,
                                    'Fire':1,
                                    'Grass':2,
                                    'Water':3}[x],y), #I'm also m
apping Normal->0, Fire->1 etc
                    )

    return X_flat,y_2

X_flat,y_2 = preprocess_data(X,y)

X_train, X_test, y_train, y_test = train_test_split(X_flat,y_2,
                                                    test_size = 0.3
3,
                                                    random_state=1,
                                                    stratify = y_2)
```

1. Test train split

In [9]:
```
X_train, X_test, y_train, y_test = map(np.array, [X_train, X_tes
t, y_train, y_test])
```

Just a note: when I split the data into training and test datasets, I make sure to do so evenly across all classes -- this is called a stratified split. The reason I do this is that I might only have like 20 grass pokemon in all of my available data. So if I split the training dataset randomly, there's a possibility that all grass pokemon end up in the test dataset (unlikely but possible). This would suck, because obviously my model isn't going to know what a grass Pokemon looks like if it's never seen a grass Pokemon before. The stratified split splits evenly across all classes (so the training dataset has representative fractions of the different Pokemon types), and therefore avoids this.

In [10]:
```
#so now we have a training dataset
print (f"we have {len(X_train)} training samples")
print (f"we have {len(X_test)} test samples")
```

```
we have 235 training samples
we have 117 test samples
```

Applying a first model

My workflow for doing practical machine learning problems is to start with a super super simple model, and work out the error (for whatever loss function we choose). Then design a slightly more complex model, and then evaluate the error and see if it's better. This is good from a psychological perspective, because you feel like you're always making progress.

Building gradually more complicated models is also good because in a real problem I will have almost definitely made a mistake somewhere in setting the problem up -- choosing loss function / downloading the data / processing the data. This is more obvious if you use a very simple model first -- complex neural networks can fail for lots of reasons, but a linear regression will only fail if you suck, or if the underlying relationship between input and output is complex or non-monotonic.

2. Choosing a model

Logistic Regression

You don't actually need any theoretical understanding for the classifiers from Sklearn to work. The default settings are pretty great almost all the time. But I feel I should at least describe some of the theory about what's going on.

Logistic regression is a simple classifier. If we just had a binary problem (i.e. the outputs were class 0 and 1), the output would be the probability of being class 1 is:

$$p(\text{class } 1) = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + ...)}}.$$

This is a function called a logit, which ranges between 0 and 1. θ_0, θ_1 etc. are the elements of $\vec{\theta}$, the model parameters. Here, the x_i are features. In this problem x_i correspond to pixel values at specific locations. Our image is 40x40x3 (last is rgb), so there are 4800 different input parameters, x_1,x_2,...,x_{4800}. This means there are 4801 different model parameters. But there is more than one type of Pokemon! There are lots of types. So we generalize this logistic regression model a "One-vs-Rest" method. That is to say that if there are N classes (here, there are four, Fire, Water etc.), we train N classifiers, each corresponding to a class. The fire classifier classifies the probability that the input is fire rather than not fire; the water classifier classifies the probabiliity that the input is water rather than not water etc. Then when running the classifier on the test dataset, the model takes in the features (i.e. the image), applies each classifer to the image, and spit out the probability that it's water rather than not water, the probability it's fire rather than not fire etc. So we end up with four numbers being spat out of the machine. Each of the four numbers should correspond to the probability that the image is that particular class, so they should sum to one. Therefore each of the individual class probabilities is then normalized by the sum of the outpus of the classifiers to ensure this is true. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

Also, what loss function are we going to use? We *could* use the fraction of misclassified examples. But our model outputs a probability distribution over all the classes for each value of X. So it makes sense to use a loss function which encodes this, punishing the model for being confidently wrong, while rewarding it for being confidently right.

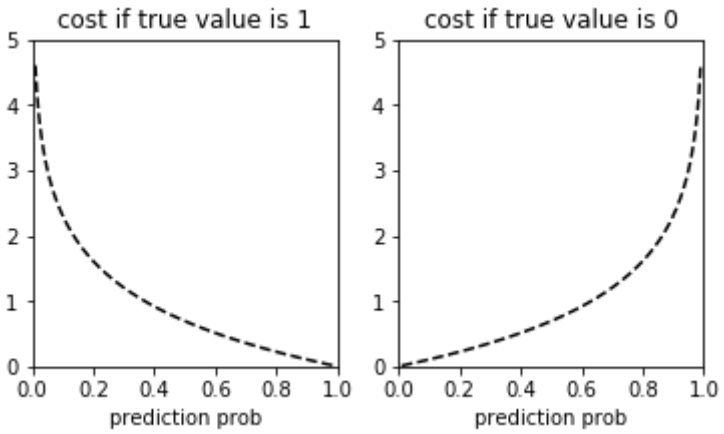
The particular quantity we choose to optimize is called the log-loss, which we optimize for each individual pair. For each pairwise logistic regression classifier (say between Fire, and Normal), the log loss is given by: The log loss function is given by:

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \log(p(y_i)) + (1-y_i)\cdot\log(1-p(y_i))$$

Here, there are N samples, and the true value of the ith sample is y_i. In this particular regression, the true values of y are either 0 (i.e. is Normal), or 1 (i.e. is Fire). Our classifier outputs a probability that (given the input), the output is Fire, and this is given by $p(y_i)$. This function looks a bit arcane, but it's got some interesting properties. Log loss is minimized (at 0), if we're fully confident in our predictions, and they're all right. But if we are ever fully confident (with probability 1 or 0), and we're wrong, our log loss diverges. So the function teaches the classifier to be conservative in predictions. Elementwise log loss functions are graphed below, demonstrating that if the model is fully confident and correct, log loss is 0, and if fully confident and incorrect, log loss is ∞. (Incidentally, the true loss function sklearn uses includes a term for regularization to prevent overfitting, https://towardsdatascience.com/regularization-in-machine-learning-76441ddcf99a) More precise details on the log-loss function are given on: https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a and here: http://neuralnetworksanddeeplearning.com/chap3.html

In [11]:
```
f,ax = plt.subplots(1,2,figsize = (6,3))
x = np.linspace(1/100.,1-1/100.,1000)
ax[0].plot(x,-np.log(x),'k--')
ax[1].plot(x,-np.log(1-x),'k--')
for a in ax: a.set_xlim(0,1) ; a.set_ylim(0,5)
ax[0].set_title('cost if true value is 1')
ax[1].set_title('cost if true value is 0')

ax[0].set_xlabel('prediction prob')
ax[1].set_xlabel('prediction prob')
```

Out[11]:
```
Text(0.5, 0, 'prediction prob')
```



4. Training parameters on the training dataset

Scikit-learn is a Python package with many classifiers / regressors / general infrastructure for machine learning. To use a model, import it, create an instantiation, and then apply the method fit, as shown below.

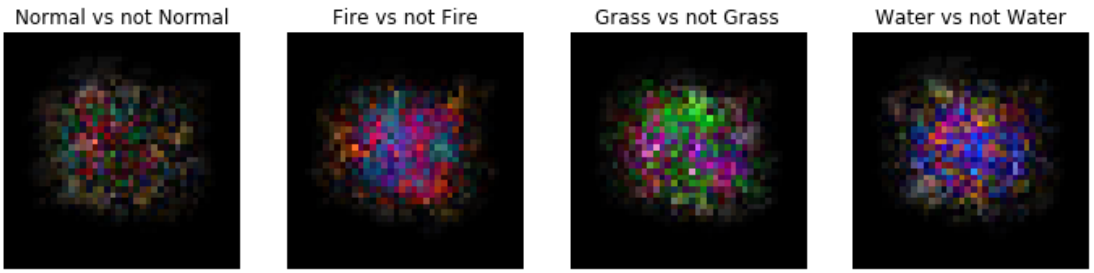
In [12]:
```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(C = 0.01, max_iter = int(1e7), multi_clas
s = "ovr")
#Hi beautiful people who are actually reading the code; to preven
t overfitting I used a MASSIVE amount of l2 regularization
#which penalizes the loss function bases the (squared) coefficien
ts themselves, i.e. \vec{\theta}\cdot\vec{\theta}.
#This prevents overfitting
#you can read more about regularization here: https://towardsdata
science.com/regularization-in-machine-learning-76441ddcf99a

lr.fit(X_train,y_train)
```

Out[12]:
```
LogisticRegression(C=0.01, class_weight=None, dual=False, fit_int
ercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=1
0000000,
                   multi_class='ovr', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001,
verbose=0,
                   warm_start=False)
```

Now we have fit the model to the training data, let's have a look at the values of $\vec{\theta}$. Remember, since we have 4 classes, we have 4 different classifiers. Also, since theta has the same dimensionality* at each of the images, we can interpret the features of the classifier as like pixels. Below I've plotted each of these, plotting the absolute value of the pixel. (I do this, because components of $\vec{\theta}$ can go negative, and it's not obvious how this is plotted...).

*Actually, there's one more parameter in the model than the input data, but we'll leave that alone for now..

In [13]:
```
f,ax = plt.subplots(1,4,figsize = (12,3))


for a,i,title in zip(ax,lr.coef_,['Normal','Fire','Grass','Water
']):
    a.imshow(np.abs(i.reshape(-1,40,3))/np.max(np.abs(i)) * 1)
    a.axis('off')
    a.set_title(f'{title} vs not {title}')
```



You can see that the logistic regression algorithm has learned to look in the center of the images; the outside is all black, which means those components of $\vec{\theta}$ are zero, which indicates the model doesn't care about them at all. You can also see that for fire, the model is sensitive to red pixels and blue pixels; for grass, it is sensitive to green pixels; for water it is sensitive to blue pixels. Normal is a bit all over the place. If you're the kind of person who finds linear algebra the easiest way to think about science what goes on is the following. The classifier takes the input image, and does the dot product with each of the four $\vec{\theta}$ values, which I've depicted as images above. If you want you can think of this as being a dot product between images. Then it throws that output (which is now a scalar), into the nonlinear "logit" function, also adding its extra value of θ_0, which you can loosely think about being related to the Baysian prior for being that class (but which isn't actually).

$$\frac{1}{1+e^{-(\vec{\theta}\cdot\vec{x}+\theta_0)}}$$

. For completeless, the technical name for this extra value of θ_0 is called the intercept.

5. Evaluate performance on the test dataset

Scikit learn models also have the method "predict", where a trained model predicts the output corresponding to its arguments

In [14]:
```
#in sample error
y_train_pred = lr.predict_proba(X_train)

from sklearn.metrics import log_loss

log_loss(y_train,y_train_pred)

y_pred = lr.predict_proba(X_test)

print (f"(in sample) logistic regression log loss is {log_loss(y_
train,y_train_pred)}")
print (f"(out of sample) logistic regression log loss is {log_los
s(y_test,y_pred)}")
```

```
(in sample) logistic regression log loss is 0.8731509017898047
(out of sample) logistic regression log loss is 1.132160227218879
8
```

This system performs very well on data it's seen before (in sample) -- it's log loss is fairly low. But it performs much less well on data it hasn't seen before (out of sample). i.e. The model is learning to memorize specific input-output pairs, rather than learning to infer deep rules about what consitutes a fire type / water type etc. This is a typical failure mode you would expect if your model has thousands of parameters, like this one. This is referred to as a failure to generalize.

It's also worth just leaving a note here on what the log loss means. If we had a binary classification problem, and we were perfect, log loss would be 0. If said we had no idea ($p = 0.5$) in response to each input, then the log loss would be $-\log(0.5) \approx 0.693$. But we have 4 classes! So if everything was randomly distributed, the log loss would be $-\log(1/4) \approx 1.39$. So we need to be beating that to be better than randomly guessing. A valid question is also: what would be our log loss if we guessed according to the population distribution: i.e. if there are 5 Normal, 3 Fire, 2 Water, and 0 Grass, we could guess Normal with probability 0.5, Fire with probability 0.3, Water with probability 0.2, never guess Grass, outputting a probability vector = [0.5,0.3,0.2,0].

The log loss of this would be the (negative) entropy of the distribution, which is given as $-\sum_i p_i \log(p_i)$. I've calculated this below.

In [15]:
```
S = 0

for i in range(4):
    p = (sum(y_train==i)/len(y_train))
    print (f"p({int_to_type[i]}) is {p}")
    S += -p*np.log(p)

print(f"The negative entropy of the distribution is {S}")
```

```
p(Normal) is 0.2978723404255319
p(Fire) is 0.14893617021276595
p(Grass) is 0.22127659574468084
p(Water) is 0.33191489361702126
The negative entropy of the distribution is 1.3441820633202994
```

If you don't find it obvious that the log loss of the trivial classifier which outputs according to the distribution is the distribution entropy, that's probably a reasonable feeling. The Wikipedia page on distribution entropy is fairly helpful... https://en.wikipedia.org/wiki/Maximum_entropy_probability_distribution

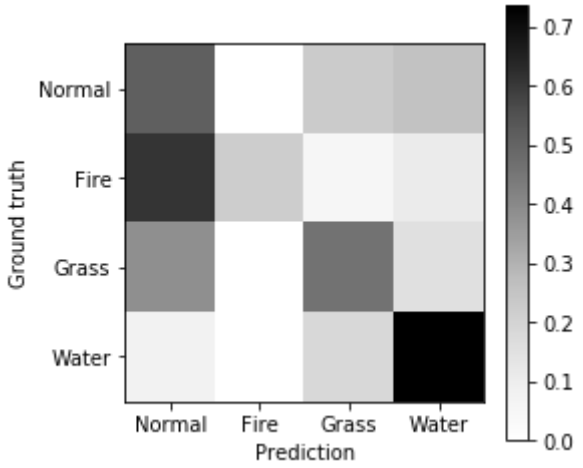
So thankfully our system also outperforms a naive classifier which outputs guesses just based on the underlying probability distribution of classes.

A visual way to look at the failures of our classification system is to use a confusion matrix. If we have N different samples, the confusion matrix row i, column j is the number of samples whose true class is i, and whose predicted class is j. So the number of Normal type which were correctly predicted is in entry (0,0) in the matrix; the number of true Normal type which are misclassified as Fire type is in entry (0,1).

In the confusion matrix below, I've divided each row of the matrix (i.e. all pokemon corresponding to a ground truth) by the sum of those pokemon. So the entry in (0,0) is the fraction of Normal Pokemon which were classified as Normal; the entry in (0,1) is the fraction of Normal Pokemon which were (mis)classified as Fire.

In [16]:
```
from sklearn.metrics import confusion_matrix
y_pred_absolute = lr.predict(X_test)
cf = confusion_matrix(y_test, y_pred_absolute)

cf_normed = []
for i in cf:
    cf_normed.append(i/np.sum(i))

f,ax = plt.subplots(1,1,figsize = (4,4))
plt.imshow(cf_normed,cmap = "Greys") #normalize...
plt.colorbar()

ax.set_xticks([0,1,2,3])
ax.set_xticklabels(['Normal','Fire','Grass','Water'])

ax.set_yticks([0,1,2,3])
ax.set_yticklabels(['Normal','Fire','Grass','Water'])

ax.set_xlabel('Prediction')
ax.set_ylabel('Ground truth')
```

Out[16]: Text(0, 0.5, 'Ground truth')



This lets us see exactly how our classifier is failing! We can see that of types which are true Normal, we're misclassifying about 20% as grass, and water, but we're (almost) never classifying normal types as fire types. You may also notice that we're classifying lots of examples as Normal. That's because there are lots of Normal Pokemon. Since there are lots of Pokemon, if you're not sure what type a Pokemon should be, Normal is a fairly decent guess, since they're represented at high frequencies in the training and test dataset. The technical name for thie problem is called, "having an unbalanced dataset" (or unbalanced classes), and there are bucketloads of ways to deal with it. https://towardsdatascience.com/handling-imbalanced-datasets-in-machine-learning-7a0e84220f28

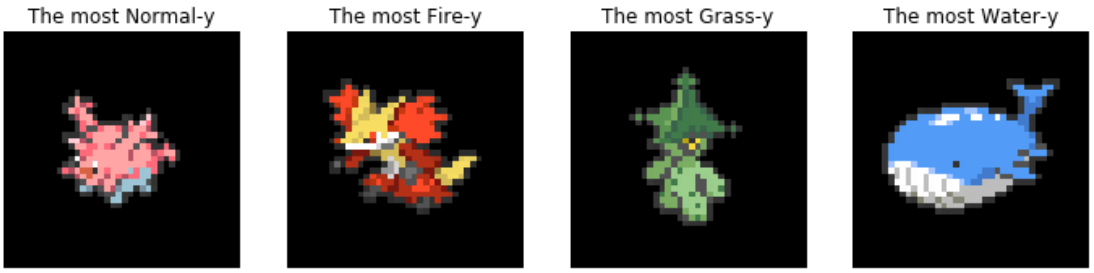
We can even find some examples for each of the 16 pairs of (true value; ML predicted value, which I've done below.

In [17]:
```
f,ax = plt.subplots(4,4,figsize = (8,8))
for i in range(4):
    for j in range(4):
        intersections = (y_pred_absolute == j) * (y_test == i) # 
true value i, but assumed value j
        if sum(intersections) != 0:
            valid_option = np.where(intersections)[0][0]
            ax[i,j].imshow(X_test[valid_option].reshape(40,40,-
3))
            ax[i,j].text(0,5,f"TRUE: {int_to_type[i]}",color = 'w
')
            ax[i,j].text(0,10,f"PRED: {int_to_type[j]}",color = '
w')
        ax[i,j].axis('off')
```

To be honest, I think our ML algorithm is doing not too badly... I think those are all reasonable mistakes which humans could have made...

Incidentally, we could now ask ourselves the question: what is the most firey Fire Pokemon, the most watery Water Pokemon, the most grassy Grass pokemon, and the most normal Normal pokemon? By this, I mean the inputs where the probabiltiy of this class is highest...

In [24]:
```
f,ax = plt.subplots(1,4,figsize = (12,3))
for i,a in enumerate(ax):
    a.imshow(X_test[np.argmax(y_pred[:,i])].reshape(40,40,3))
    a.axis('off')
    a.set_title(f"The most {int_to_type[i]}-y")
```



Looks fairly reasonable...

Can we do better?

Feature Engineering

We know that pretty obviously pokemon types are based on color palette. You can even see above in our matrix of misclassified examples that this is basically how the model we developed before is doing classification. So maybe, one way to make our classifier much better is to design a feature that fits our intuition about how type is predicted.

One obvious feature is the average pixel color across all pixels which are not blank. This consists of 3 floats, its 'redness', its 'blueness' and its 'greeness'

In [80]:
```
def make_average_pixel_feature(x):
    x_reshaped = x.reshape(1600,-3)
    loc_0 = np.where(np.sum(x_reshaped,axis = 1) != 0)
    mean_pixels = np.mean(x_reshaped[loc_0],axis = 0)
    return mean_pixels
```

In [81]:
```
X_train_single_pixel = np.array([make_average_pixel_feature(x) fo
r x in X_train])
X_test_single_pixel = np.array([make_average_pixel_feature(x) for
x in X_test])
```

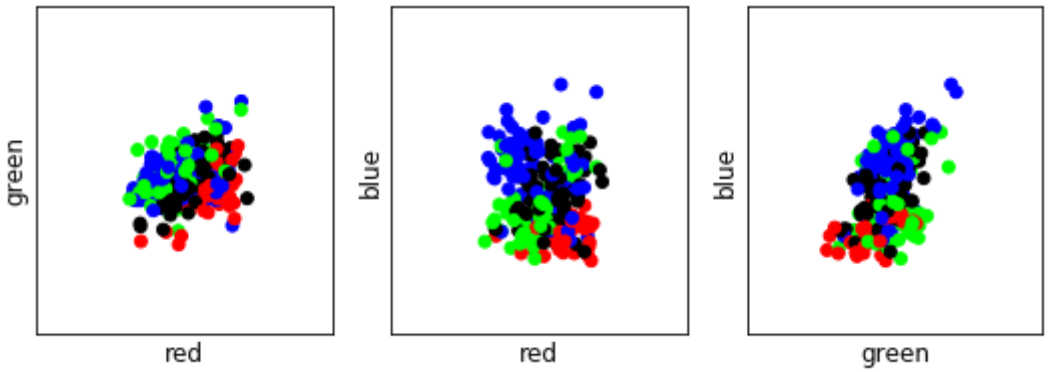
Now we've manually reduced the dimensionality to 3D, we can see if there are obvious clusters, (as we might expect)

Below, I have plotted the single average pixel for each of element in the training dataset. Color codes are red for fire, green for grass, blue for water, and black for normal

In [82]:
```
colors = list(map(lambda i : {0:[0,0,0], 1:[1,0,0],2:[0,1,0],
3:[0,0,1]}[i],  y_train))
```

In [83]:
```
f,ax = plt.subplots(1,3,figsize = (9,3))
ax[0].scatter(X_train_single_pixel[:,0],X_train_single_pixel[:,
1],c = colors,)
ax[1].scatter(X_train_single_pixel[:,0],X_train_single_pixel[:,
2],c = colors,)
ax[2].scatter(X_train_single_pixel[:,1],X_train_single_pixel[:,
2],c = colors,)

for a in ax:
    a.set_xlim(0,1)
    a.set_ylim(0,1)
    a.set_xticks([])
    a.set_yticks([])

ax[0].set_xlabel('red') ; ax[0].set_ylabel('green')
ax[1].set_xlabel('red') ; ax[1].set_ylabel('blue')
ax[2].set_xlabel('green') ; ax[2].set_ylabel('blue')
```

Out[83]:
```
Text(0, 0.5, 'blue')
```



It seems that our average pixel approach doesn't obviously delineate the classes; it would be difficult to draw a boundary between Fire and Water etc. We can still try applying a logistic regression.

In [84]:
```
lr_engineered = LogisticRegression()

lr_engineered.fit(X_train_single_pixel,y_train)
```

Out[84]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_inte
rcept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=1
00,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001,
verbose=0,
                   warm_start=False)
```

In [85]:
```
y_train_pred = lr_engineered.predict_proba(X_train_single_pixel)

in_sample_log_loss = log_loss(y_train,y_train_pred)

y_pred = lr_engineered.predict_proba(X_test_single_pixel)
y_pred_absolute = lr_engineered.predict(X_test_single_pixel)


print (f"(in sample) logistic regression log loss is {in_sample_l
og_loss}")
print (f"(out of sample) logistic regression log loss is {log_los
s(y_test,y_pred)}")
```

```
(in sample) logistic regression log loss is 1.15318016564041
(out of sample) logistic regression log loss is 1.178361336514503
1
```

Interesting! Our logistic regression over the average pixel has performed about the same as our logistic regression over all pixels! (recall that our log loss for the logistic regression over all pixels was 1.13). Also, it has generalized fairly well (i.e. our in sample error is comparable to our out of sample error, so our model has actually learned underlying features of the data, rather than just memorized examples! Typically, high dimensional systems (we have 4800 dimensions here) with few training examples see good performance when a domain specific expert designs features accoring to intuition, and domain specific understanding.

Ways you can extend this

Here, we've only touched the basics of ML prediction systems. If you want to improve the performance of our model further, there are lots of things you can play with: 1. One could add additional hand designed features to the single pixel classification system. Maybe add the sum of RGB pixels to the logistic regression as a feature? 1. Use a systematic dimensionality reduction tool, like PCA to reduce the number of free parameters in the image https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html 1. For the logistic regression over all pixel values, you could artificiallly expand the dataset, by including rotations and reflections https://www.kdnuggets.com/2020/02/easy-image-dataset-augmentation-tensorflow.html 1. You can add other types ("Dark", "Electric"), and see how well your model performs. 1. Identify problematic examples, whose ground truth is one type, but which the classifier confidently predicts are another.