INFO5002: Intro to Python for Info Sys

Week 12

Slides created by: Zachary Doucet

Week 12

I. Regularisation

II. Logistic Regression

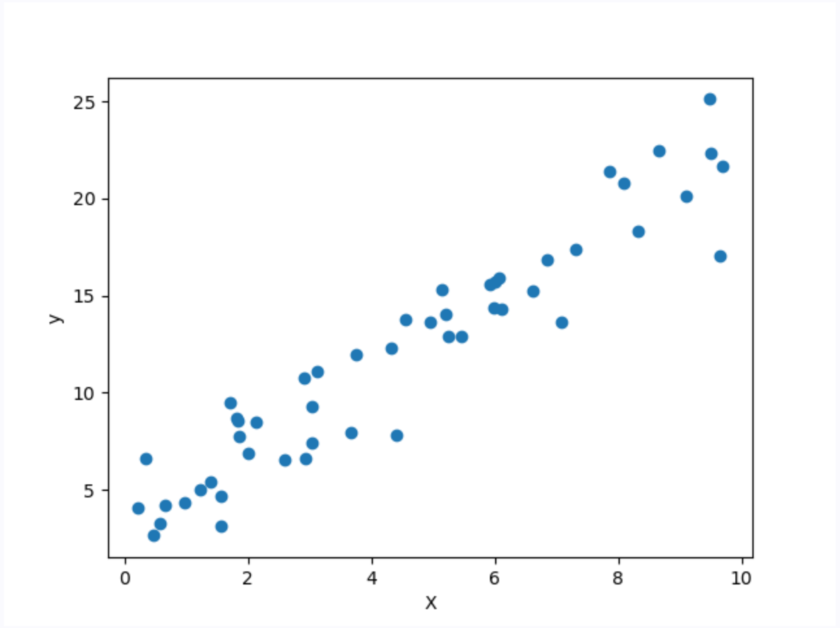
III. Decision Trees

Review

Basics of ML

- What a model is.
- How to collect and break data for training.
- How to evaluate binary classification.
- How to find a closed-form solution.
- How to find an open-form solution for gradient descent.

Linear regression



$$y=m*x+b$$

```python
def predict(x, m, b):
    return m * x + b
```

$$\frac{\partial J}{\partial m}=-\frac{1}{n}\sum x_i(y_i-m*x_i-b)$$

$$\frac{\partial J}{\partial b}=-\frac{1}{n}\sum y_i-m*x_i-b$$

Multiple Regression

- When you have more than one feature.

$$\hat{y} = a + \underset{\underset{\text{Feature 1}}{\uparrow}}{\beta_1} * x_1 + \beta_2 * x_2 + \ldots + \underset{\underset{\text{Feature k}}{\uparrow}}{\beta_k} * x_k$$

$$\nabla_\beta J = -\frac{1}{n} X^T (y - \hat{y}) \qquad \hat{y} = X\beta$$

Regularisation

DSfS 200-201

What you may encounter

1. Overfitting as you use more features.
2. Complexity in understanding the model as you use more features.

Regularization

- Technique that **punishes** the larger the weights are. Excluding the constant.
  - Ridge Regression: penalty proportional to the sum of the squares of the weights.
  - Lasso Regression: penalty proportional to the sum of absolute weights.

Ridge Regression

$$ridge = \sum \beta_i^2$$

$$\nabla_\beta ridge = \left\{ \begin{matrix} \beta_0 \\ 2\alpha\beta_1 \\ \dots \\ 2\alpha\beta_d \end{matrix} \right\}$$

Ridge coefficient

Results

```
R^2: 0.6836432580624692
Features:
- Base: 4.672601277149279
- crim: -0.15711983546272634
- zn: 0.038691170336494225
- indus: 0.017075882217822292
- chas: 1.2524062926091726
- nox: 1.459931933862347
- rm: 4.782462447638699
- age: 0.001123772489480381
- dis: -0.7628326393125271
- rad: 0.20204237541852388
- tax: -0.007919294822269869
- ptratio: -0.32660600167865195
- black: 0.006932403221789192
- lstat: -0.5240291117290363
```

```
R^2: 0.4262929416047012
Features:
- Base: 0.0038808773033996847
- crim: -0.09846100594513182
- zn: 0.06071704842293404
- indus: 0.0548379972900202
- chas: 0.07778463840639091
- nox: 0.1208480722824499
- rm: 2.0428558245834174
- age: 0.06346593521553487
- dis: -0.36749548163088236
- rad: 0.1341659389097766
- tax: 0.001558819237903661
- ptratio: 0.22002054138271926
- black: 0.014885991236595288
- lstat: -0.6611153308384818
```

Lasso Regression

$$lasso = \sum |\beta_i|$$

This derivative is a bit difficult.

Essentially each feature is +1, -1, or [-1, 1]

Logistic Regression

DSfS 203-214

What if we want our output

- To be a probability or between [0, 1]?
- Our simple and multiple linear regression can give us arbitrarily large and small values.
- We need to use a special function.

Logistic Function

$$logistic(x) = \frac{1}{1+e^{-x}}$$

```python
def logistic(x):
    return 1.0 / (1 + math.exp(-x))
```

- As x increases, e^-x gets smaller => closer to 1.
- As x decreases, e^-x gets bigger => closer to 0.

Convenient derivative

$$\frac{d}{dx} logistic(x) = logistic(x) * (1 - logistic(x))$$

```
def logistic_grad(x):
    y = logistic(x)
    return y * ( 1 - y)
```

Logistic Regression

$$\hat{y} = f(X\beta)$$

Where f is the logistic function

```
def predict(X, beta):
   m = np.matmul(X, beta)
   return logistic(m)
```

Instead of minimising error

- We can maximise the likelihood.

$$L(y|X,\beta)=f(X\beta)^{y}(1-f(X\beta))^{1-y}$$

$$\log L(y|X,\beta)=y\log f(X\beta)+(1-y)\log(1-f(X\beta))$$

- Since we use gradient descent we will minimise the negative log likelihood.

Oh gradients

$$\frac{\partial}{\partial \beta_j} -L(y|x,\beta) = -(y - f(\underset{\substack{\uparrow \\ \text{Vector (not matrix)}}}{x} \cdot \beta)) * x[j]$$

$$\nabla_\beta -L(y|x,\beta) = \begin{Bmatrix} -(y - f(x\cdot\beta)) * x[0] \\ -(y - f(x\cdot\beta)) * x[1] \\ \dots \\ -(y - f(x\cdot\beta)) * x[d] \end{Bmatrix}$$

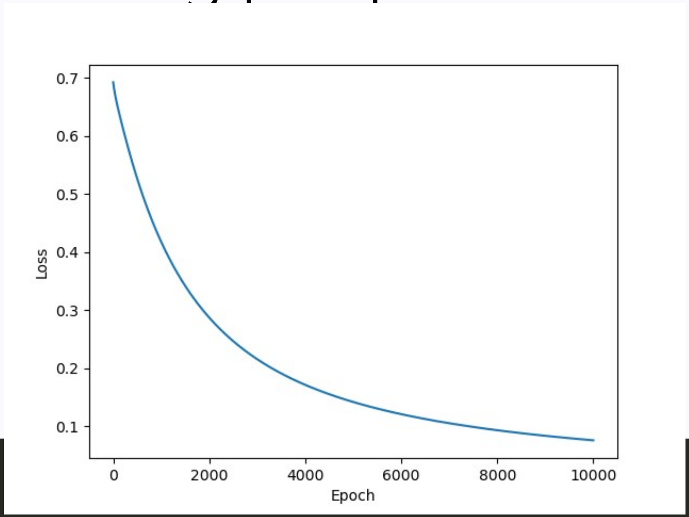
Independent samples

- Assuming each sample is independent then the overall likelihood is just the multiplication of each.
  - Then for log-likelihood it is the addition.

$$\nabla_{\beta} - L(y|X,\beta) = \begin{pmatrix} -(y-f(x\cdot\beta)) * x_{0,0} \\ -(y-f(x\cdot\beta)) * x_{0,1} \\ \dots \\ -(y-f(x\cdot\beta)) * x_{0,d} \end{pmatrix} + \dots + \begin{pmatrix} -(y-f(x\cdot\beta)) * x_{n,0} \\ -(y-f(x\cdot\beta)) * x_{n,1} \\ \dots \\ -(y-f(x\cdot\beta)) * x_{n,d} \end{pmatrix}$$

Loss as negative log likelihodd

- Training on Iris dataset where “setosa” is 0 and “versicolor” is 1, and using pos prob of 0.5—I get 100% accuracy.



The positive probability is the cutoff where if predicted >= pos prob then we evaluate as 1.

```
def loss(self, X, y):
    y_pred = self.predict(X)
        epsilon = 1e-7  # Avoid log(0)
        return -np.mean(y * np.log(y_pred + epsilon) +
            (1 - y) * np.log(1 - y_pred + epsilon))
```

Recap

- When we have a binary decision then logistic regression is the solution.
- Make sure to set an appropriate pos prob threshold.

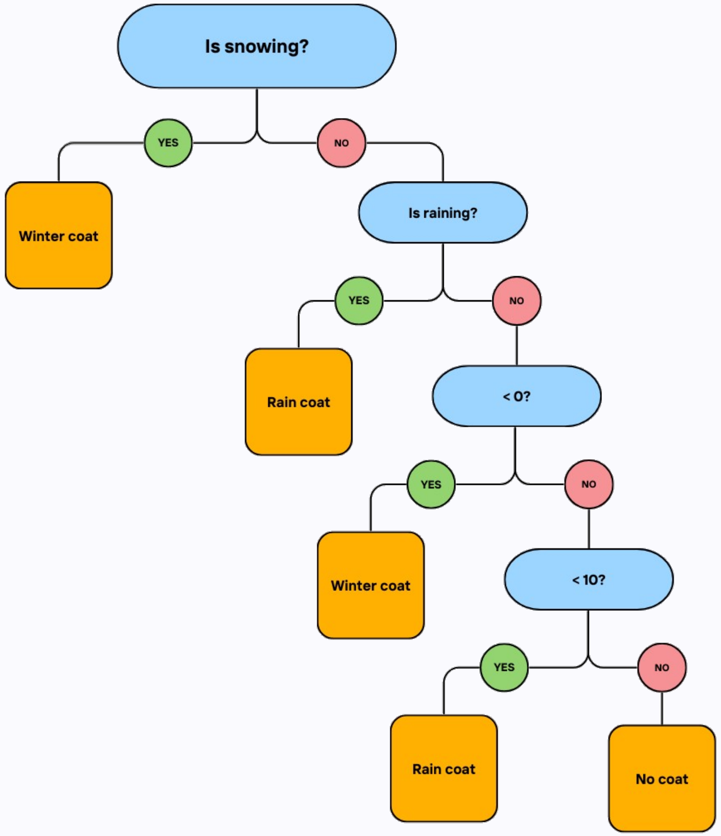
Decision Trees

DSfS 215-225

How do we make decisions?

- Usually we start by looking at one variable and then based on some condition you either do something or another thing.
- If it is snowing, I would wear a winter jacket; if it is raining, a rain jacket; if it is neither but below 0C, I would wear a winter jacket; if below 10C, a rain jacket; if not anything, then no jacket.

As a decision tree



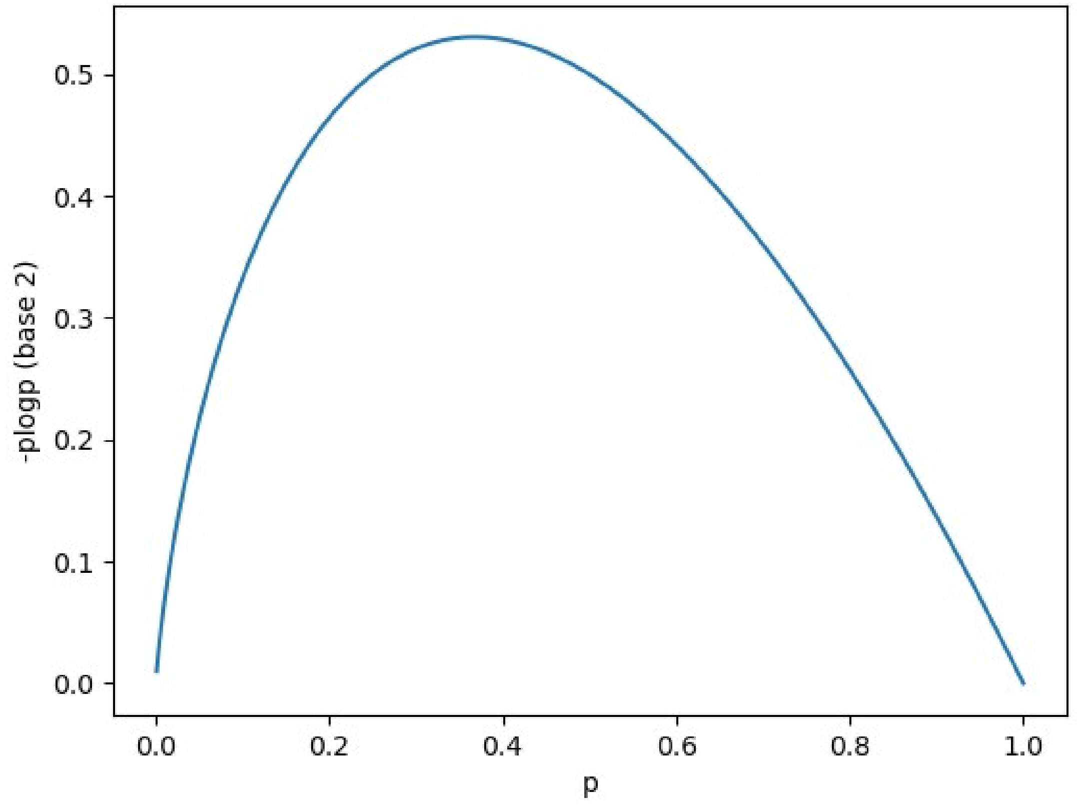
But not manually!

- What if we create a learned algorithm to create these decision trees and create decisions by itself?!
- Nice because then we can give the decisions to a human (human readable), to evaluate.
- Works on numeric and categorical features. No need to numerically encode.

We want good decisions.

- Good decisions are able to split up possibilities in a balanced way. Think of Akinator.
- We can use entropy to represent how spread the data is. p_i is proportion of class_i. The more spread, the less certain.

$$H(s) = -p_1 \log_2 p_1 - \ldots - p_n \log_2 p_n$$



```
def entropy(ps):
    return np.sum(-ps * np.log2(ps))
```

To know how good a split is

- We can weighted sum the entropies of each subset.

  Where q_i is the proportion of samples in S_i.

$$H = q_1 H(S_1) + \ldots + q_m H(S_m)$$

- Smaller is better.

ID3

- If data all have same label => create leaf node of that label.
- If list of attributes (what to split on) is empty => create leaf node that predicts the most popular label.
- If not empty try to partition by each attribute.
  - Choose the one with lowest entropy.
  - Add decision node based on attribute.
  - Recurse on each partitioned subset on remaining attributes.

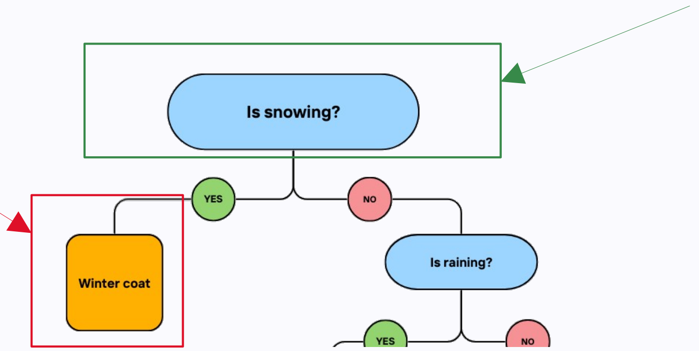
Basic data structures

```python
class Leaf:
    def __init__(self, val):
        self.val = val
```

Represents our possible output

```python
class Node:
    def __init__(self, key):
        self.key = key
        self.children = {}
```

Represents our question we hold



```python
def build_tree(self, data, attributes):
    if len(data) == 0:
       return self.default # no data
    if len(data.unique()) == 1:
       return Leaf(data["label"][0]) # all the same (lowest entropy)
    if len(attributes) == 0:
        return Leaf(data["label"].mode())

    entropies = [self.entropy_partition_by_attribute(data, attribute)
         for attribute in attributes]
    min_idx = entropies.index(min(entropies))
    best_attribute = attributes[min_idx]
    children = self.split_by_attribute(data, best_attribute)
    node = Node(best_attribute) # create a new decision node
    new_attributes = [attribute for attribute
         in attributes if attribute != best_attribute] # remove what we used
   # recurse over each child
   for child in children:
        node.children[child] = self.build_tree(children[child], new_attributes)

   return node
```

Numerical hack

- Given that two numbers have infinite points between them you can subdivide the max – min by a certain amount as your possible classes.

Overview

- Decision trees are nice because they are easily explainable and human readable.
- They can work with both categorical and numerical values.
- They struggle with overfitting.

Let’s practice

- On the Iris dataset (on Canvas) try to train both a Logistic Regression and a Decision Tree using scikit-learn LogisticRegression and scikit-learn DecisionTreeClassifier.
- Choose any two of the flower species and assign one as 0 and the other as 1. In fact for Decision Tree’s you do not even need to do the mapping (still keep the same two species for consistency).

Perceptron

DSfS 227-229