Introduction to Data Science
Lecture 8
Machine Learning 1

EMSE 6992 Fall 2019

Ben Harvey

Outline

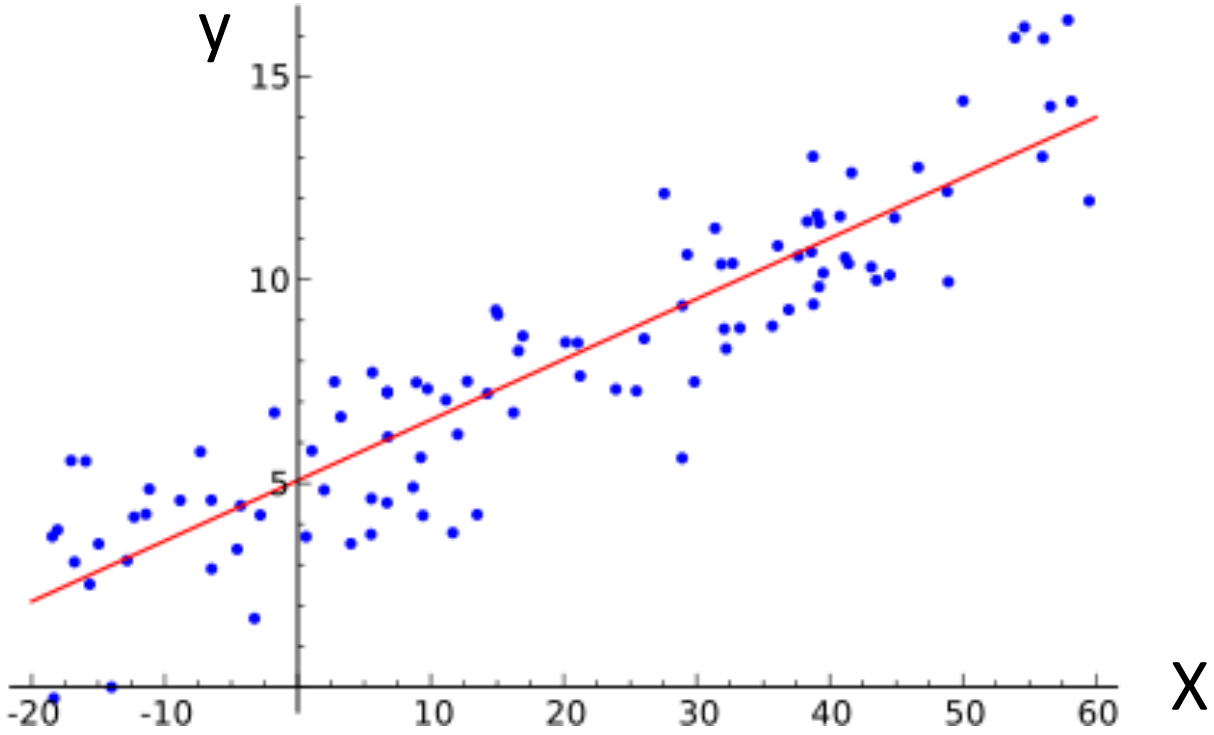
- **Supervised Learning**
  - **Linear Regression**
    - R-squared
  - Logistic Regression
  - SVMs
- Gradient-based optimization
- Decision Trees and Random Forests
- HW Examples

Machine Learning

- **Supervised:** We are given input samples (X) and output samples (y) of a function y = f(X). We would like to "learn" f, and evaluate it on new data. Types:
  - **Classification:** y is discrete (class labels).
  - **Regression:** y is continuous, e.g. linear regression.

Linear Regression

We want to find the “best” line (linear function y=f(X)) to explain the data.



Linear Regression

The predicted value of y is given by:

$$\hat{y} = \hat{\beta}_0 + \sum_{j=1}^{p} X_j \hat{\beta}_j$$

The vector of coefficients β is the regression model.

Linear Regression

The regression formula

$$\hat{y} = \hat{\beta}_0 + \sum_{j=1}^{p} X_j \hat{\beta}_j$$

if $X_0 = 1$, can be written as a matrix product with X a row vector:

$$y = \mathrm{X}\beta$$

We get this by writing all of the input samples in a single matrix **X**:

i.e. **rows of X** $= \begin{pmatrix} X_{11} & \cdots & X_{1n} \\ \vdots & \ddots & \vdots \\ X_{m1} & \cdots & X_{mn} \end{pmatrix}$

are **distinct observations**, **columns of X** are **input features**.

Least Squares Solution

The most common measure of fit between the line and the data is the **least-squares fit**.

There is a good reason for this: If the points are generated by an ideal line with additive Gaussian noise, the least squares solution is the ***maximum likelihood solution***.

Probability of a point y_j is $\Pr(y_j) = \exp\left(\frac{-(y_j - X_j\beta)^2}{2\sigma^2}\right)$ and the probability for all points is the product over j of $\Pr(y_j)$.

We can **easily maximize the log** of this expression $\frac{-(y_j - X_j\beta)^2}{2\sigma^2}$ for one point, or the sum of this expression at all points.

Iterative Least Squares

The exact method requires us to invert a matrix $(\mathbf{X}^T\mathbf{X})$ whose size is M^2 for M features and takes time $O(M^3)$. This is **too big** for large feature spaces like text or event data.

Gradient methods reduce the SS error using the **derivative wrt** β

$$\mathrm{RSS}(\beta) = \sum_{i=1}^{N} (y_i - \beta x_i)^2$$

which is

$$\nabla = \mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta)$$

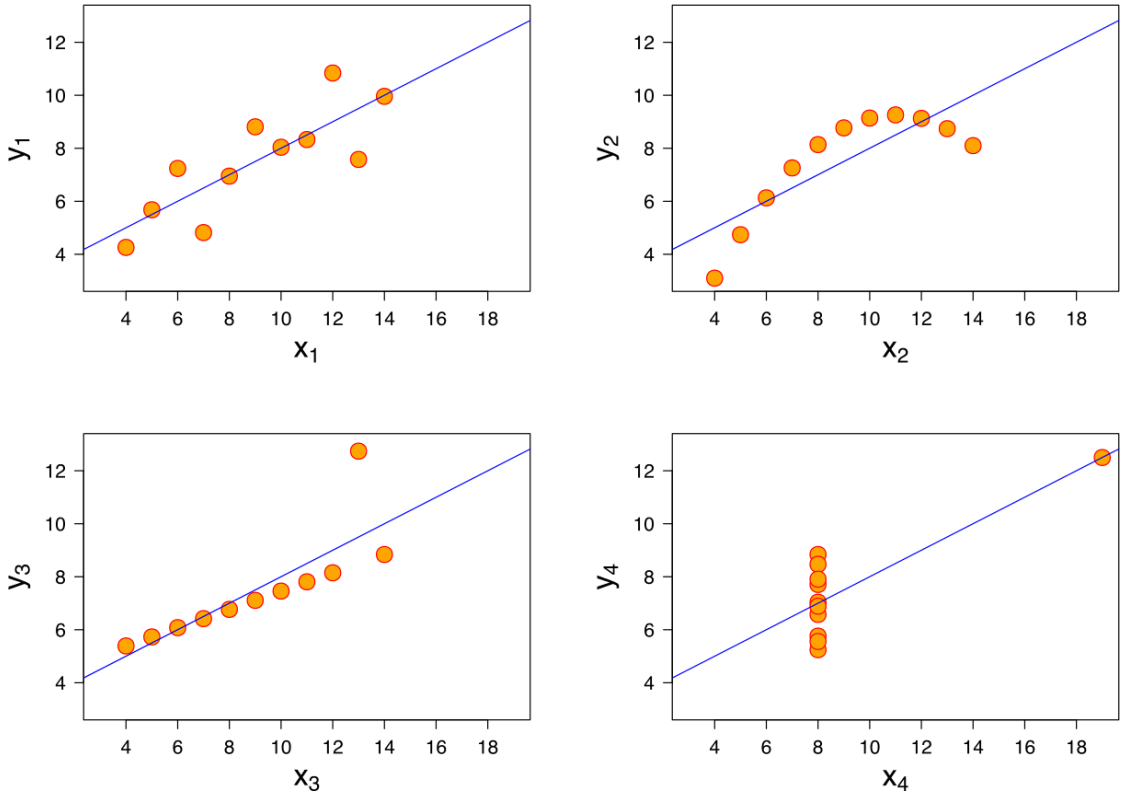
More on this later in this lecture.

Outline

- Supervised Learning
  - Linear Regression
    - **R-squared**
  - Logistic Regression
  - SVMs
- Gradient-based optimization
- Decision Trees and Random Forests
- HW 2 Examples

R^2-values and P-values

We can **always** fit a linear model to any dataset, but how do we know if there is a **real linear relationship**?



R^2-values

Approach: Measure how much the total "noise" (variance) is reduced when we include the line as an offset.

R-squared: a suitable measure. Let $\hat{y} = X\hat{\beta}$ be a predicted value, and y be the sample mean. Then the R-squared value is

$$R^2 = 1 - \frac{(y_i - \hat{y}_i)^2}{(y_i - \bar{y})^2}$$

And can be described as the fraction of the total variance not explained by the model.

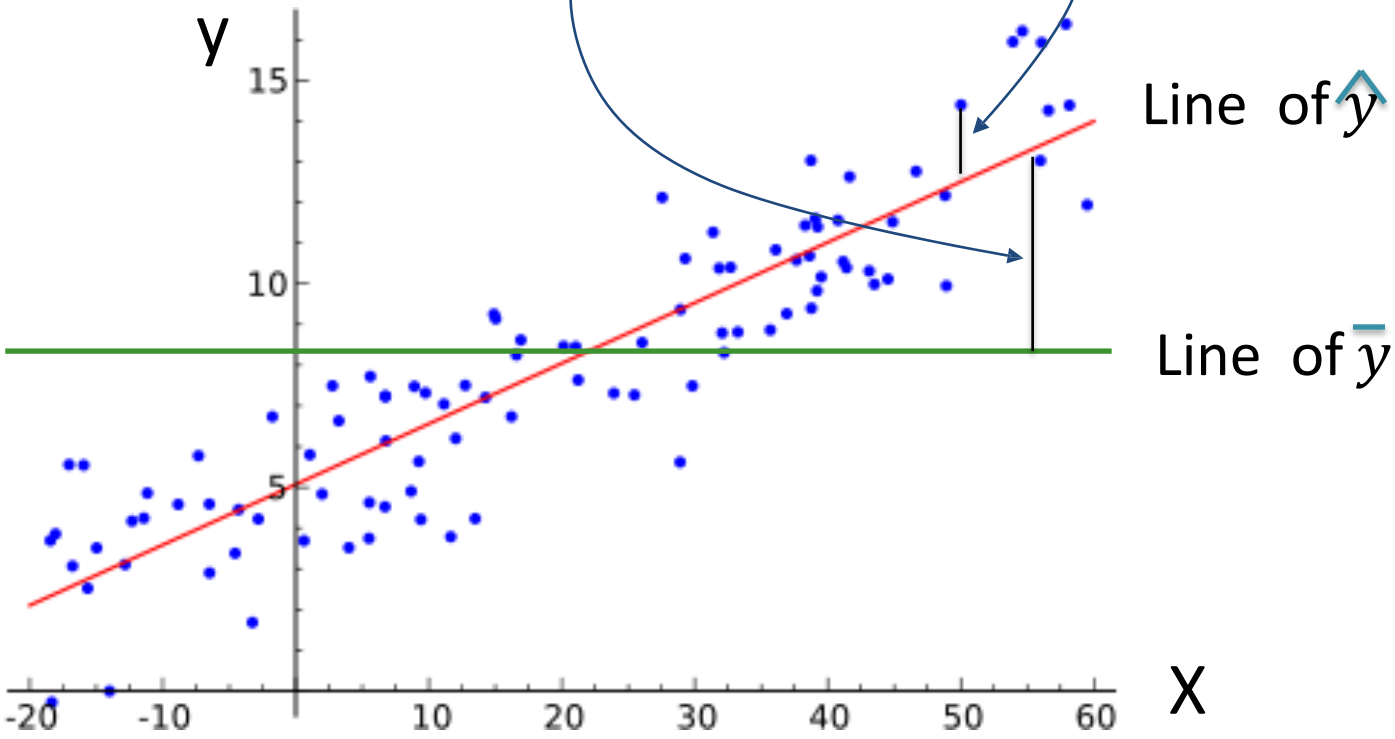
$R^2 = 0$: bad model. No evidence of a linear relationship.

$R^2 = 1$: good model. The line perfectly fits the data.

R-squared

Small if good fit

$$R^2 = 1 - \frac{(y_i - \hat{y}_i)^2}{(y_i - \bar{y})^2}$$



R^2-values and P-values

Statistic: From R-squared we can derive another statistic (using degrees of freedom) that has a standard distribution called an **F-distribution**.

From the CDF for the F-distribution, we can derive a **P-value** for the data.

The P-value is, as usual, the probability of observing the data under the null hypothesis of no linear relationship.

If **p is small**, say less than 0.05, we conclude that **there is a linear relationship**.

Outline

- Supervised Learning
  - Linear Regression
    - R-squared
  - **Logistic Regression**
  - SVMs
- Gradient-based optimization
- Decision Trees and Random Forests
- HW 2 Examples

Logistic Regression

- We made a distinction earlier between regression (predicting a real value) and classification (predicting a discrete value).

- Logistic regression is designed as a **binary classifier** (output say {0,1}) but actually **outputs the probability** that the input instance is in the "1" class.

- A logistic classifier has the form:

$$p(X) = \frac{1}{1 + \exp(-X\beta)}$$

where $X = (X_1, \ldots, X_n)$ is a vector of features.

Logistic Regression

- Logistic regression is probably the **most widely used general-purpose classifier**.

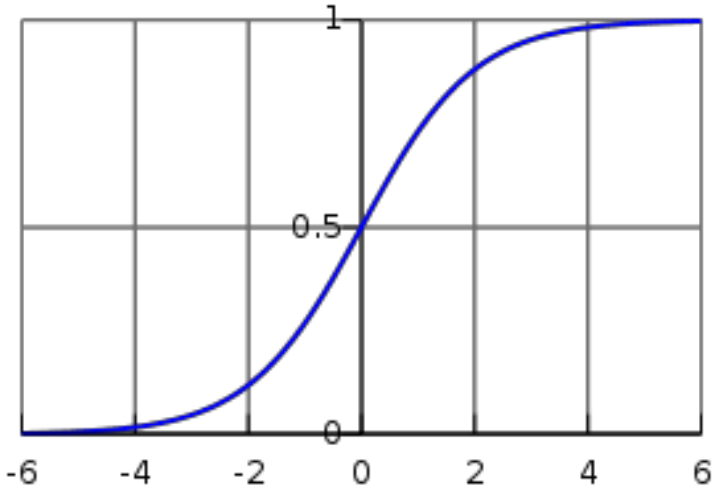
- Its **very scalable** and can be **very fast** to train. It's used for
  - Spam filtering
  - News message classification
  - Web site classification
  - Product classification
  - Most classification problems with large, sparse feature sets.

- The only caveat is that **it can overfit** on very sparse data, so its often used with Regularization

Logistic Regression

- Logistic regression maps the "regression" value $-X\beta$ in $(-\infty,\infty)$ to the range [0,1] using a "logistic" function:

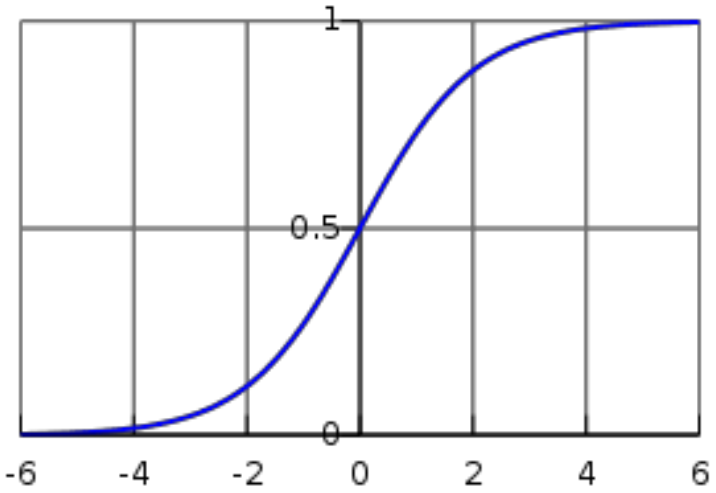
$$p(X) = \frac{1}{1+\exp(-X\beta)}$$



- i.e. the logistic function maps any value on the real line to a probability in the range [0,1]

Logistic Regression

- Where did the logistic function come from?



Logistic Regression and Naïve Bayes

- Logistic regression is actually **a generalization of Naïve Bayes with binary features**.
- Logistic Regression can model a Naïve Bayes classifier when the binary features are **independent**.
- Bayes rule for two classes c and $\neg c$:

$$\Pr(c|X) = \frac{\Pr(X|c)\Pr(c)}{\Pr(X)} = \frac{\Pr(X|c)\Pr(c)}{\Pr(X|c)\Pr(c) + \Pr(X|\neg c)\Pr(\neg c)}$$

- Dividing by the numerator:

$$= \frac{1}{1 + \frac{\Pr(X|\neg c)\Pr(\neg c)}{\Pr(X|c)\Pr(c)}}$$

Logistic Regression and Naïve Bayes

We have

$$\Pr(c|X) = \frac{1}{1 + \frac{\Pr(X|\neg c)\Pr(\neg c)}{\Pr(X|c)\Pr(c)}} = \frac{1}{1 + \exp(-X\beta)}$$

which matches if

$$\frac{\Pr(X|\neg c)\Pr(\neg c)}{\Pr(X|c)\Pr(c)} = \exp(-X\beta)$$

and assuming feature independence, the two sides factor as:

$$\frac{\Pr(\neg c)}{\Pr(c)} \prod_{i=1}^{n} \frac{\Pr(X_i|\neg c)}{\Pr(X_i|c)} = \exp(-\beta_0) \prod_{i=1}^{n} \exp(-X_i \beta_i)$$

And we can match corresponding (i) terms to define β.

Logistic Regression and Naïve Bayes

Summary: Logistic regression has this form:

Models Naïve Bayes formula with two classes
after dividing through by one of them

$$\Pr(c|X) = \frac{1}{1 + \exp(-X\beta)}$$

Models product of contributions
from different (independent) features

Logistic Regression and Naïve Bayes

- Because it can always learn an NB model but is more general, Logistic regression should **do at least as well as** naïve Bayes*
- Logistic regression **typically does better** though because it can deal with **dependencies** between features, whereas Naïve Bayes cannot.
- - true **if** we have sufficient data to reduce variance in the models.
- Naive Bayes generally has **higher bias** due to the independence assumption, but also **lower variance** than logistic regression.

Logistic Training

For training, we start with a collection of **input values** X^i and corresponding **output labels** $y^i \in \{0,1\}$. Let p^i be the **predicted output** on input X^i, so

$$? = \frac{1}{1 + \exp(X^i \beta)}$$

The **accuracy** on an input X^i is

$$A^i = y^i p^i + (1 - y^i)(1 - p^i)$$

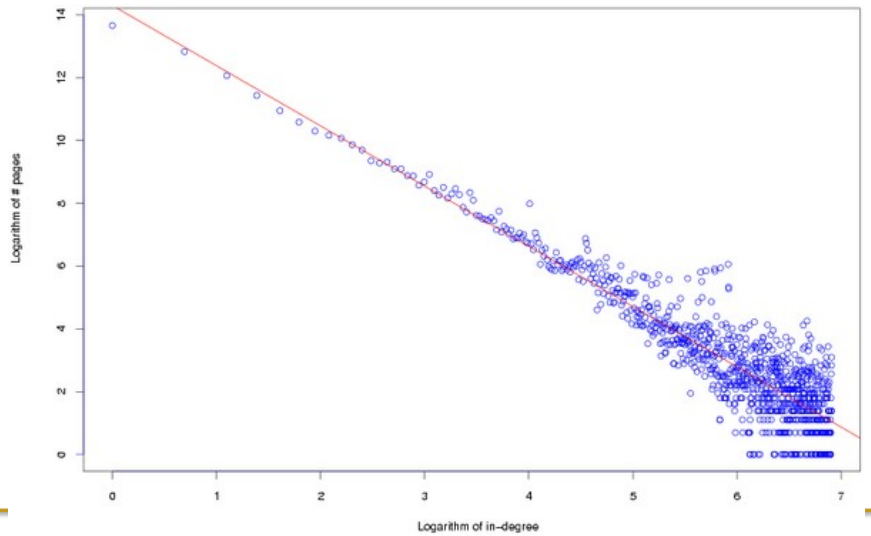
Logistic regression maximizes either the sum of the log accuracy, or the total accuracy, e.g.

$$A = \sum_{i=1}^{N} \log A^i$$

Logistic Regression Training - SGD

- A very efficient way to train logistic models is with **Stochastic Gradient Descent** (SGD) – we keep updating the model with gradients from small blocks (mini-batches) of input data.

- One challenge with training on power law data (i.e. most data) is that the terms in the gradient can have very different strengths (because of the power law distribution). We'll discuess this soon…

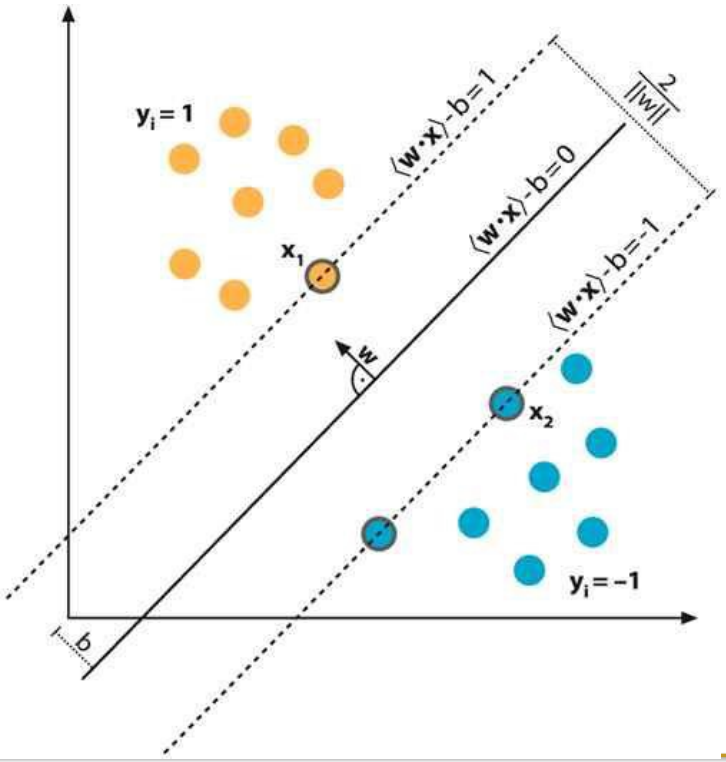


Outline

- Supervised Learning
  - Linear Regression
    - R-squared
  - Logistic Regression
  - **SVMs**
- Gradient-based optimization
- Decision Trees and Random Forests
- HW 2 Examples

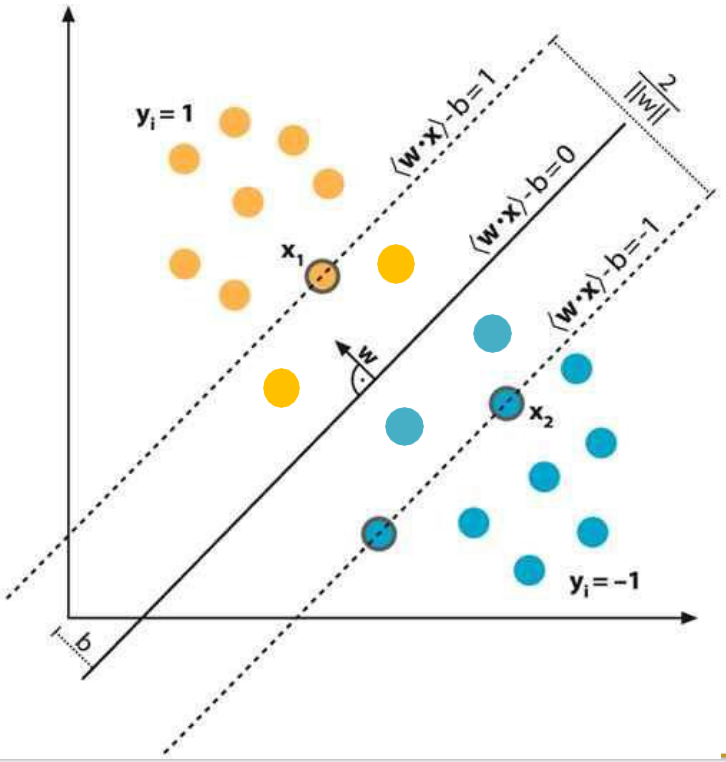
Support Vector Machines

- A Support Vector Machine (SVM) is a classifier that tries to **maximize the margin** between training data and the classification boundary (the plane defined by $X\beta = 0$)



Support Vector Machines

- The idea is that maximizing the margin **maximizes the chance that classification will be correct on new data**. We assume the new data of each class is near the training data of that type.



SVM Training

SVMs can be trained using SGD.

The **SVM gradient can be defined** as (here $p^i = X^i\beta$)

$$\frac{dA}{d\beta} = \sum_{i=1}^{N} \text{if } \left(p^i y^i < 1\right) \text{ then } y^i X^i \text{ else } 0$$

The expression $\left(p^i y^i < 1\right)$ tests whether the point X^i is nearer than the margin, and if so adds it with sign y^i. This "nudges" the model to push it further out next time. It ignores other points.

SVM Training

This SGD training method (called Pegasos) is much faster than previous methods, and competitive with Logistic Regression.

Its also capable of training in less than one pass over a dataset.

Outline

- Supervised Learning
  - Linear Regression
    - R-squared
  - Logistic Regression
  - SVMs
- **Gradient-Based Optimization**
- Decision Trees and Random Forests
- HW2 Examples

Newton's Method

Newton's method is the classical approach to iterative optimization. In vector/matrix form it is:

$$X^1 = X^0 - H_f^{-1}(X^0)\, G_f(X^0)$$

Where $G_f = \frac{\partial f}{dX_i}$ is the gradient and

$H_f = \frac{d^2 f}{dX_i dX_j}$ (a matrix) is the Hessian.

Newton's method converges very fast when its feasible. But there are several challenges to using it with large datasets.

Challenges for Newton's Method

1. Both the gradient and Hessian are normally computed **on the entire dataset**. This means many passes over the data to complete a series of Newton steps.

2. The Hessian has size $O(M^2)$ if there are M features. This is impractical for large feature spaces, e.g. text or event data.

One Solution is **L-BFGS**, which stands for "Limited Memory Broyden-Fletcher-Goldfarb-Shanno". It's a Newton-like algorithm that use an approximation to the Hessian with only a few rows.

L-BFGS still requires full passes over the dataset however.

Stochastic Gradient

A very important set of iterative algorithms use **stochastic gradient** updates.

They use a **small subset or mini-batch X** of the data, and use it to compute a gradient which is added to the model

$$\beta' = \beta + \alpha \nabla$$

Where α is called the **learning rate**.

These updates happen **many times** in one pass over the dataset.

Its possible to compute high-quality models with very few passes, sometime with less than one pass over a large dataset.

Challenges for Stochastic Gradient

Stochastic gradient has some serious limitations however, especially if the **gradients vary widely in magnitude**. Some coefficients change very fast, others very slowly. (The Hessian corrects for this in Newton's method).

This happens for **text, user activity and social media data** (and other power-law data), because gradient magnitudes scale with feature frequency, i.e. over several orders of magnitude.

Its not possible to set a single learning rate that trains the frequent and infrequent features at the same time.

ADAGRAD – Adaptive-rate SGD

ADAGRAD is a particularly simple and fast approach to this problem. The gradient g_t at the t[th] step is scaled as

$$\frac{g_t}{\sqrt{\sum_{k=1}^{t} {g_k}^2}}$$

Where ${g_k}^2$ is the k[th] gradient vector squared element-wise.

This corrects for feature scale factors, and all ADAGRAD-scaled gradient components have similar magnitudes.

ADAGRAD often improves SGD convergence on power-law data **by orders of magnitude**.

SGD learning rate schedules

With ADAGRAD gradients are scaled by

$$\frac{1}{\sqrt{\sum_{k=1}^{t} {g_k}^2}}$$

Which is proportional to $t^{-0.5}$ assuming gradient magnitudes are roughly constant. This **inverse power rate** is quite efficient for "easy" learning problems like linear or logistic regression.

SGD learning rate schedules

For harder (non-convex) learning problems, its common to use learning rate schedules that decay more slowly, or not at all.

Exponential schedule: $l = l_0 \exp(-t/\tau)$ starting with an initial learning rate l_0, the rate decreases by a factor of e for each interval of learning τ.

Constant schedule: $l = l_0$ is good for very complex learning problems, e.g. multi-layer neural networks.

Linear schedule: $l = l_0 \left(1 - t/T_{final}\right)$ where T_{final} is the final value of t. Uses a fast initial rate to move close to the final optimum, and then slows to reduce variance.

5-minute break

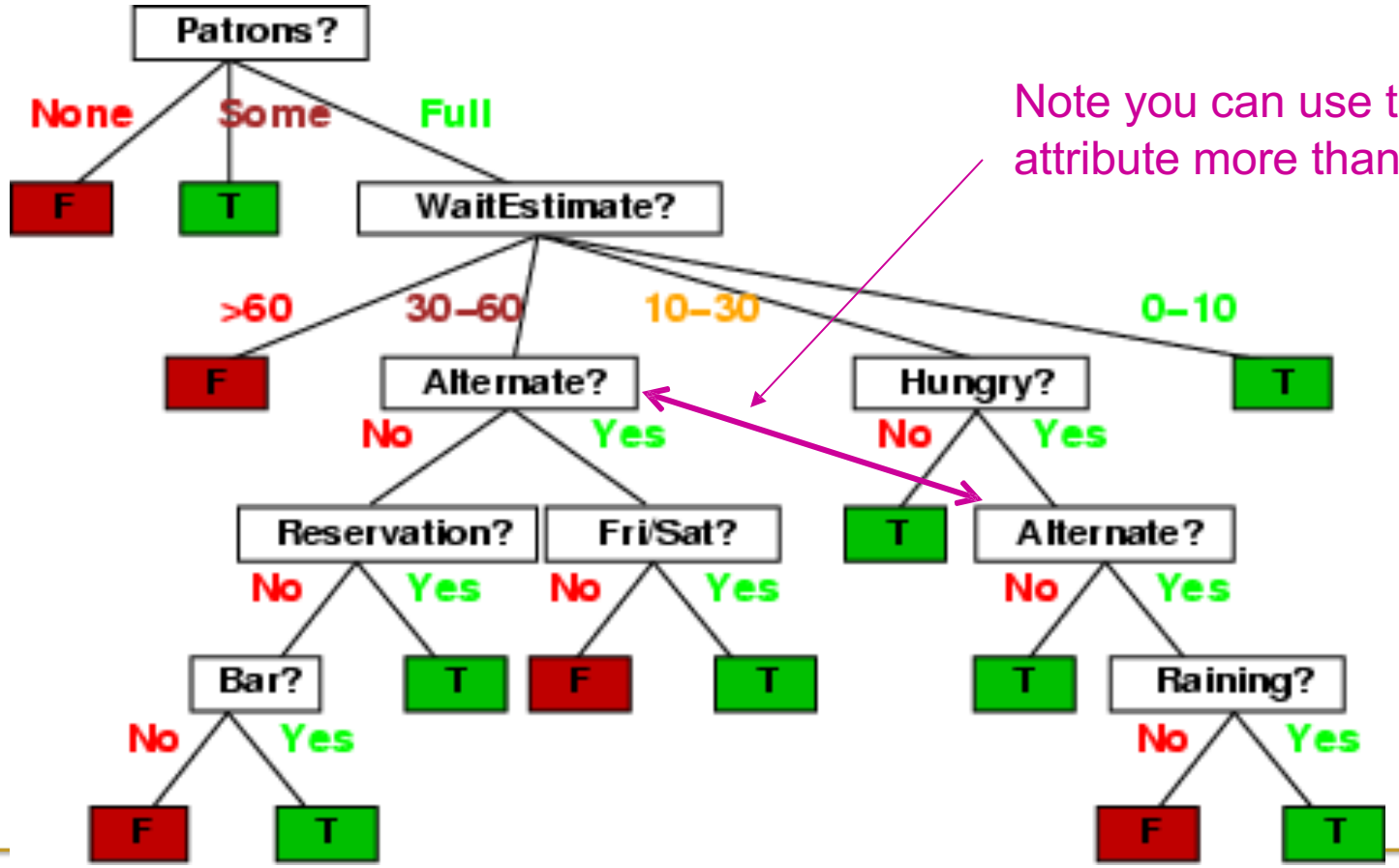
Outline

- Supervised Learning
  - Linear Regression
    - R-squared
  - Logistic Regression
  - SVMs
- Gradient-based optimization
- **Decision Trees and Random Forests**
- HW 2 Examples

Decision trees

- Walk from root to a class-labeled leaf.
- At each node, branch based on the value of some feature.

Ex: Should we wait for a table at this restaurant?

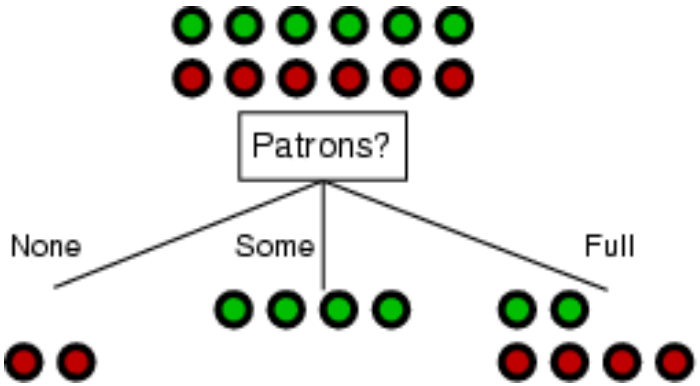


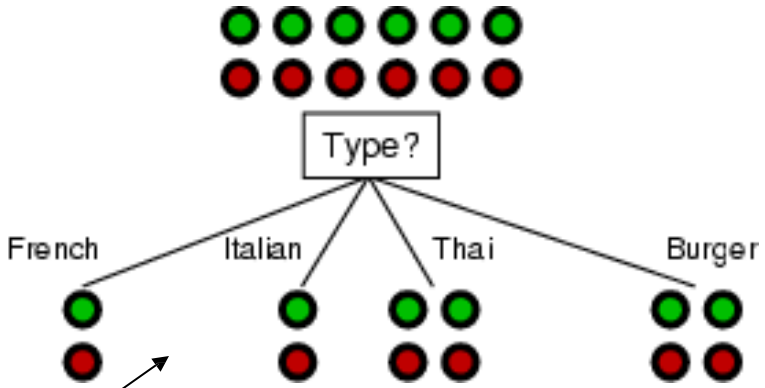
Decision tree learning

- If there are k features, a decision tree might have up to 2^k nodes. This is usually much too big in practice.
- We want to find “efficient” (smaller) trees.
- We can do this in a **greedy manner by recursively choosing a best split feature at each node**.

Choosing an attribute

- Idea: a good features splits the examples into subsets that are (ideally) "all positive" or "all negative"





- *Patrons or type?*

To wait or not to wait is still at 50%.

Using Information Theory

Entropy is defined at each node based on the class breakdown:

- Let p_i be the fraction of examples in class i.
- Let p_i^f be the fraction of elements with feature f that lie in class i.
- Let $p_i^{\neg f}$ be the fraction of elements without feature f that lie in class i

Finally let p^f and $p^{\neg f}$ be the fraction of nodes with (respectively without) feature f

Information Gain

Before the split by f, entropy is

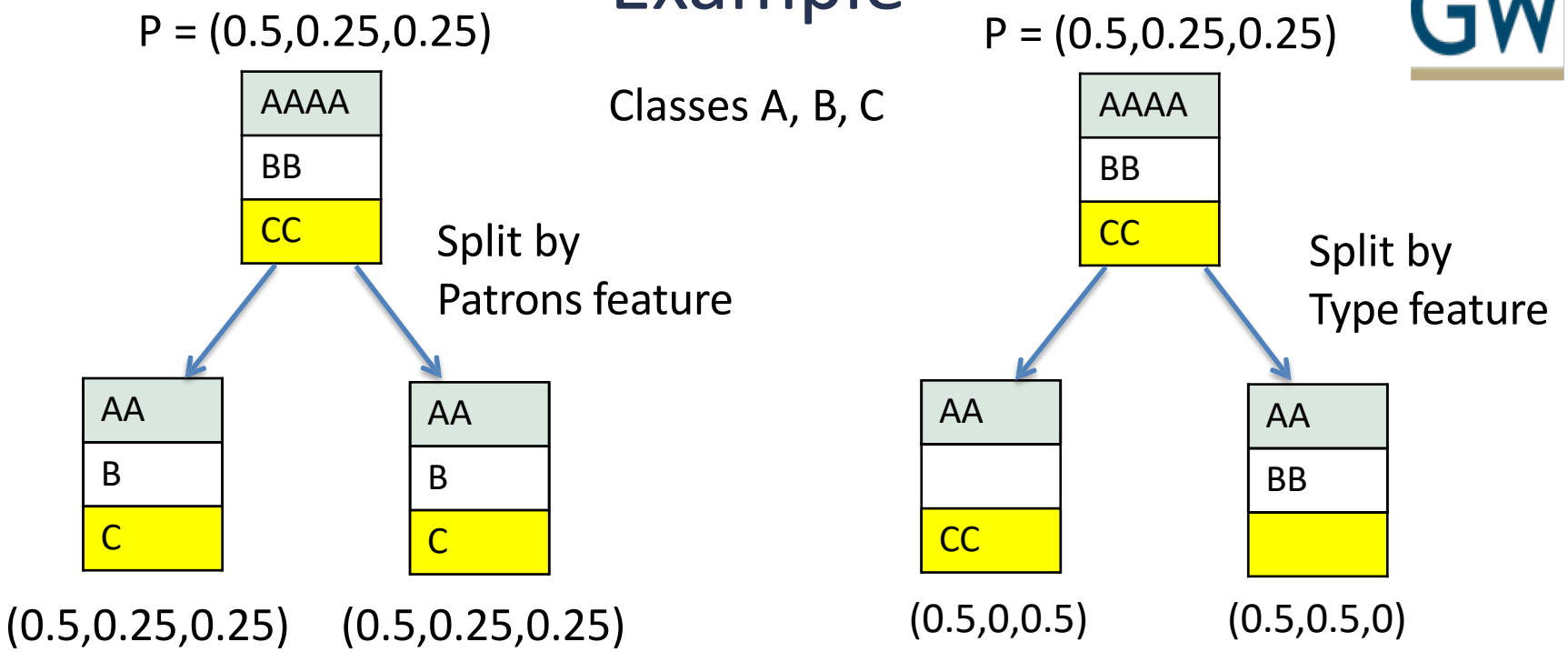
$$E = -\sum_{i=1}^{m} p_i \log p_i$$

After split by f, the entropy is

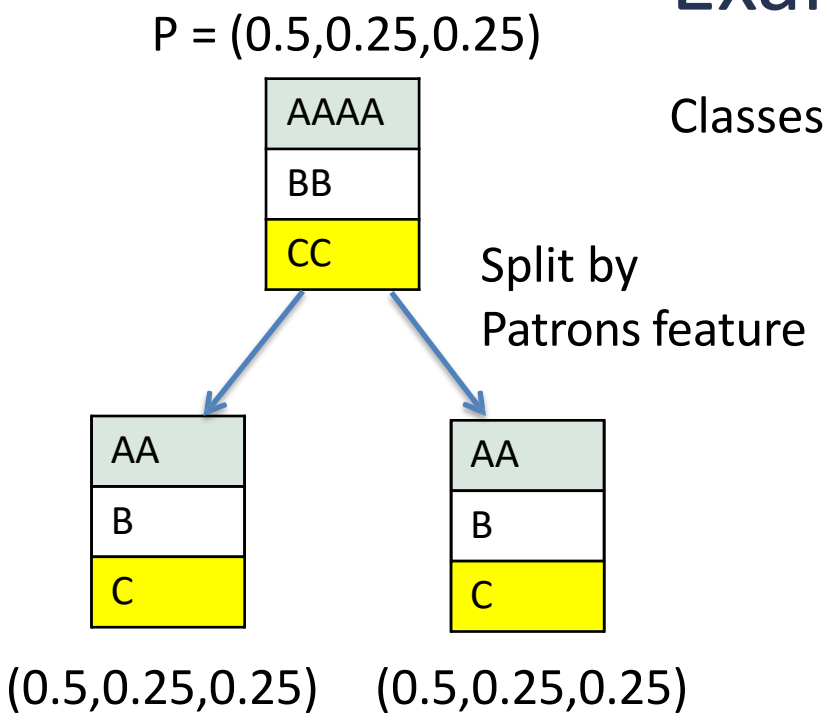
$$E_f = -p^f \sum_{i=1}^{m} p_i^f \log p_i^f - p^{\neg f} \sum_{i=1}^{m} p_i^{\neg f} \log p_i^{\neg f}$$

The information gain = $E - E_f$ (information = -entropy)

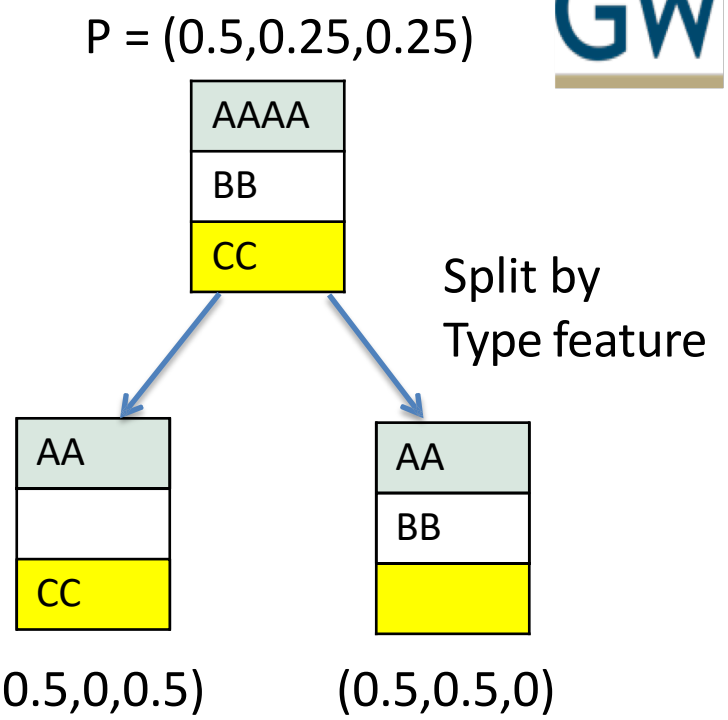
Example


P = (0.5,0.25,0.25)
AAAA
BB
CC
Split by
Patrons feature
AA
B
C
AA
B
C
(0.5,0.25,0.25)
(0.5,0.25,0.25)
Classes A, B, C
P = (0.5,0.25,0.25)
AAAA
BB
CC
Split by
Type feature
AA
CC
AA
BB
(0.5,0,0.5)
(0.5,0.5,0)

Example



Classes A, B, C



$E = -\quad p_i \log p_i =$

0.5*1+0.25*2+0.25*2 = 1.5 bits

After:

$E_f =$ (0.5+0.5)*1.5= 1.5 bits

No gain!

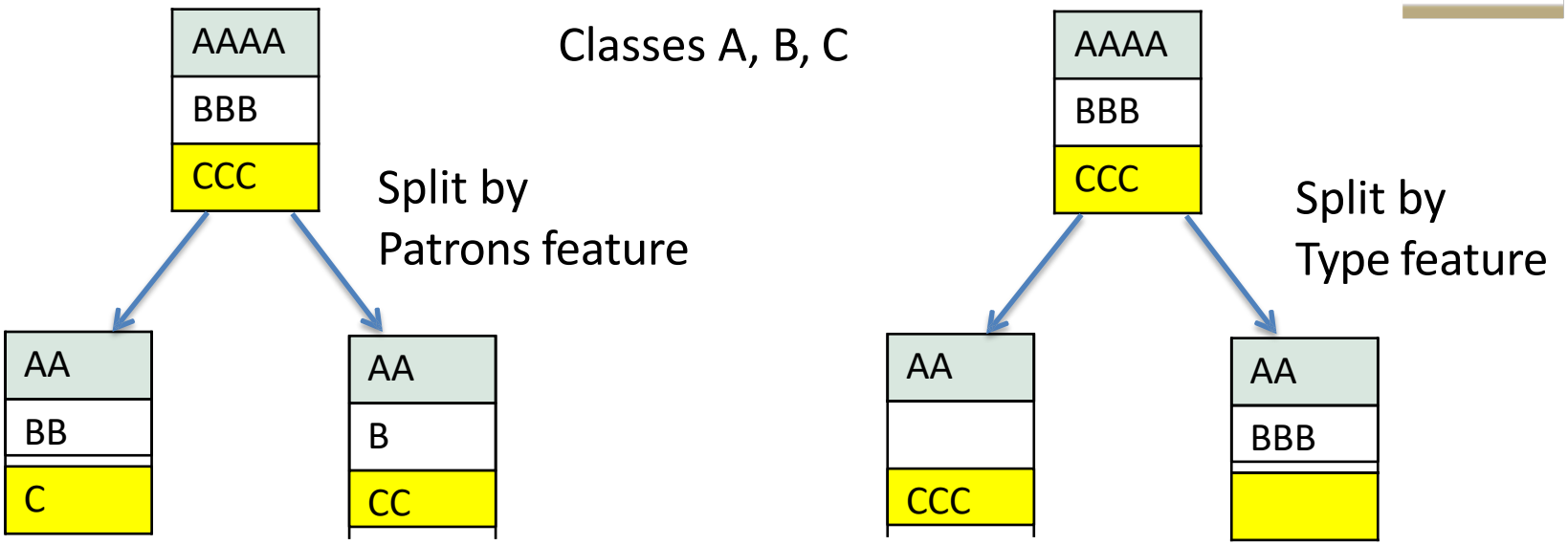
Before: E = 1.5 bits

After:

$E_f =$ (0.5+0.5)*1 bits = 1 bits

Gain = $E \ - \ E_f$ = 0.5 bits

Example


AAAA
BBB
CCC
Classes A, B, C
Split by
Patrons feature
AA
BB
C
AA
B
CC
AAAA
BBB
CCC
Split by
Type feature
AA
CCC
AA
BBB

Choosing best features

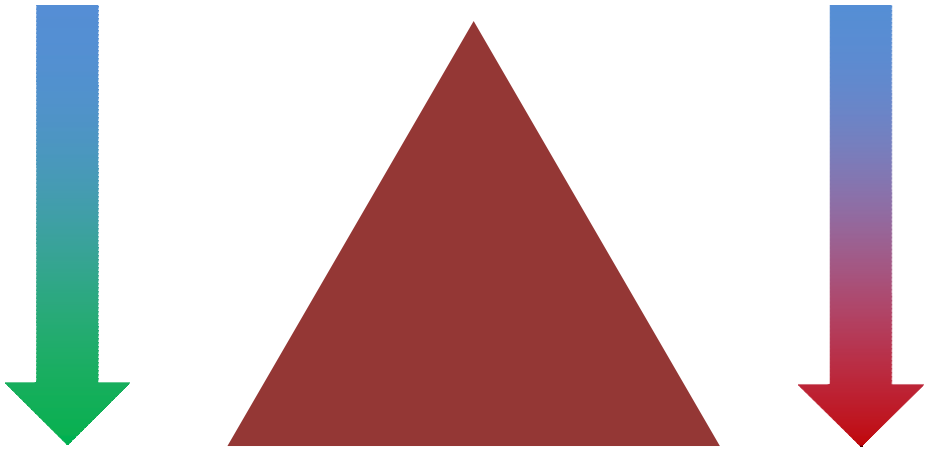
At each node, we choose the feature f which **maximizes the information gain**.

This tends to be produce mixtures of classes at each node that **are more and more "pure"** as you go down the tree.

If a node has examples all of one class c, we make it a leaf and output "c". Otherwise, when we hit the depth limit, we output **the most popular class** at that node.

Decision Tree Models

- As tree depth increases, bias decreases and variance generally increases. Why? (Hint: think about k-NN)



Bias decreases with tree depth

Variance increases with tree depth

Ensemble Methods

Are like **Crowdsourced machine learning algorithms**:

- Take a collection of simple or *weak* learners
- Combine their results to make a single, better learner

Types:

- **Bagging:** train learners in parallel on different samples of the data, then combine by voting (discrete output) or by averaging (continuous output).
- **Stacking:** combine model outputs using a second-stage learner like linear regression.
- **Boosting:** train learners on the filtered output of other learners.

Random Forests

Grow K trees on datasets sampled from the original dataset with replacement (bootstrap samples), p = number of features.

- Draw K bootstrap samples of size N
- Grow each Decision Tree, by selecting a random set of m out of p features at each node, and choosing the best feature to split on.
- Aggregate the predictions of the trees (most popular vote) to produce the final class.

Typically m might be e.g. sqrt(p) but can be smaller.

Random Forests

- Principles: we want to take a vote between different learners so we don't want the models to be too similar. These two criteria ensure diversity in the individual trees:
- Draw K bootstrap samples of size N:
  - Each tree is trained on different data.
- Grow a Decision Tree, by selecting a random set of m out of p features at each node, and choosing the best feature to split on.
  - Corresponding nodes in different trees (usually) cant use the same features to split

Random Forests

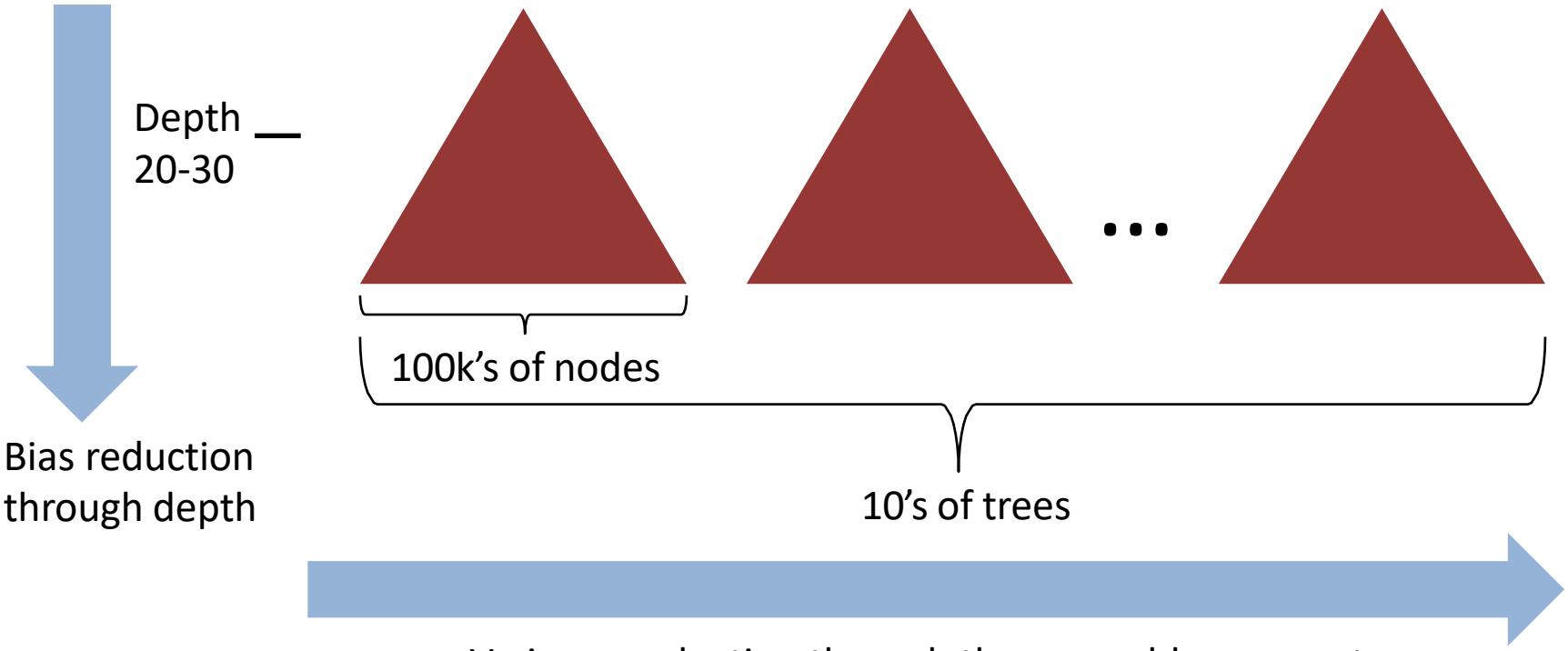
- **Very popular in practice,** probably the most popular classifier for dense data (<= a few thousand features)
- **Easy to implement** (train a lot of trees). Good match for MapReduce.
- **Parallelizes easily** (but not necessarily efficiently).
- **Not quite state-of-the-art accuracy** – DNNs generally do better, and sometimes gradient boosted trees.
- **Needs many passes over the data** – at least the max depth of the trees. (<< boosted trees though)
- **Easy to overfit** – hard to balance accuracy/fit tradeoff.

Boosted Decision Trees

- A recently-developed alternative to random Forests:
- In contrast to RFs whose trees are trained **independently**, BDT trees are trained **sequentially** by **boosting**: Each tree is trained on weighted data which emphasizes incorrectly-labeled instances by the previous trees.
- Both methods can produce very high-quality models. Superiority of one method or the other is very dataset-dependent.
- Resource requirements are very different as well, so its actually non-trivial to compare the methods (what resources do you fix during the experiment?).

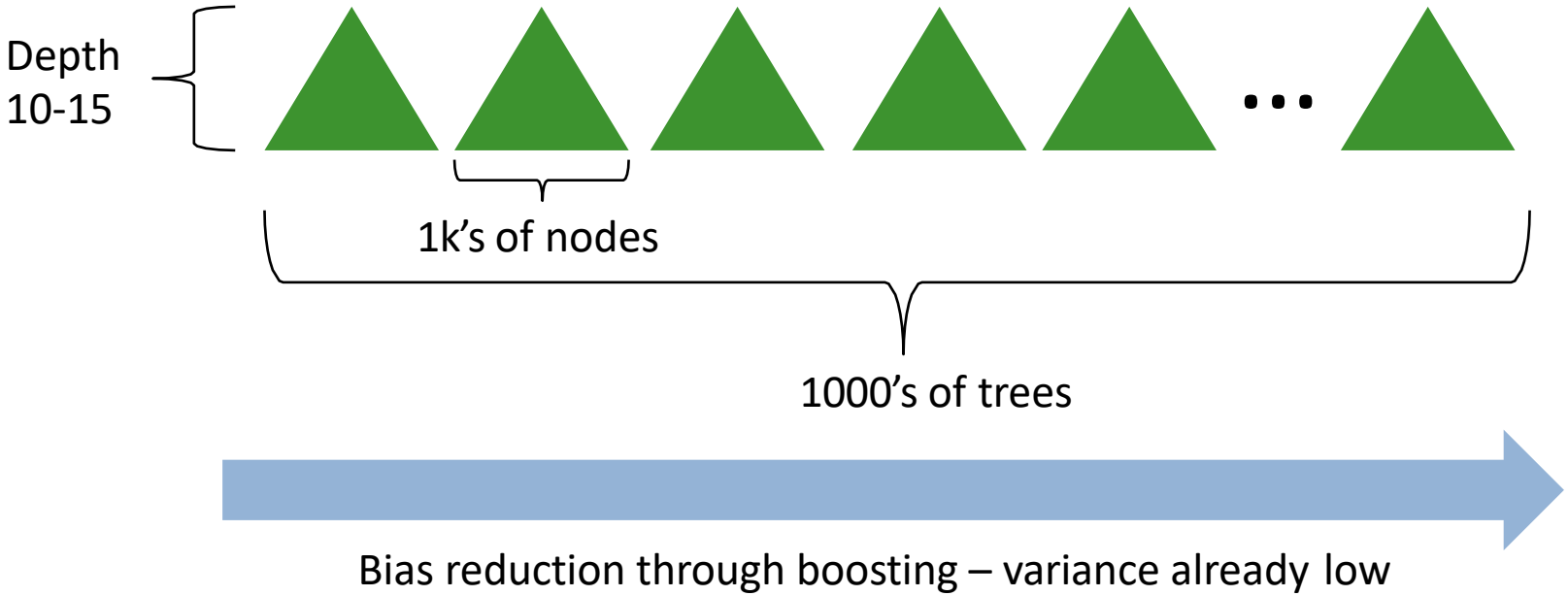
Random Forests vs Boosted Trees

- The “geometry” of the methods is very different (MNIST data):
- Random forest use 10’s of deep, large trees:



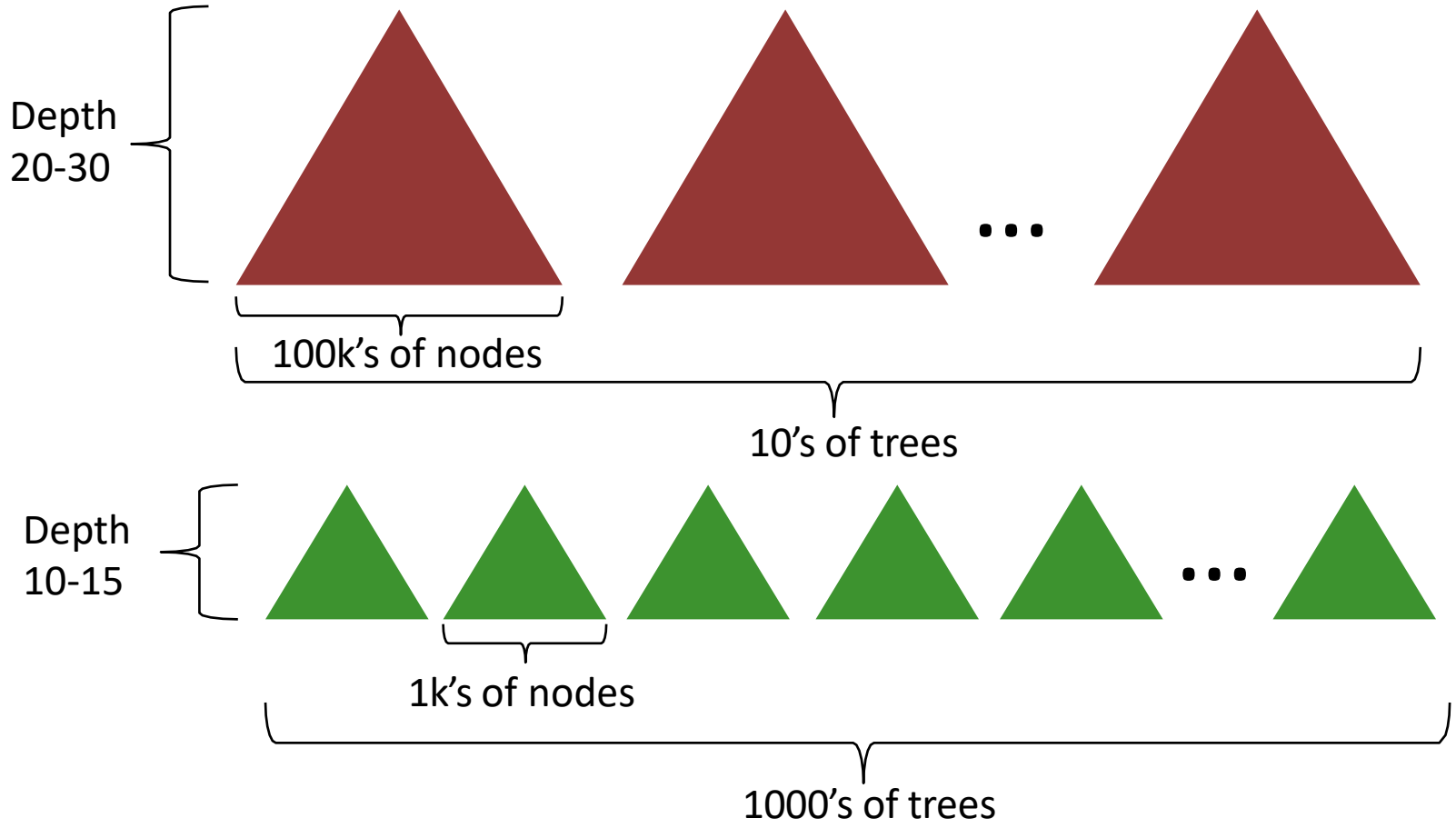
Random Forests vs Boosted Trees

- The “geometry” of the methods is very different (MNIST data):
- Boosted decision trees use 1000’s of shallow, small trees:



Random Forests vs Boosted Trees

- RF training embarrassingly parallel, can be very fast
- Evaluation of trees (runtime) also much faster for RFs



Take-aways

- Beware HiPPOs and folklore about ML methods – especially those that consider one dimension of performance:
  - Naïve Bayes vs. Logistic Regression
  - Random Forests vs. Boosted Decision Trees
- Always try out all the plausible methods whenever possible.

- Make sure you understand the principles and characteristics of the methods you are using. And make sure to tune each method thoroughly - it can make a huge difference to performance.

Model Quality

Almost every model **optimizes some quality criterion**:

- For linear regression it was the **Residual Sum-of-Squares**
- For k-Means it is the **"Inertia"** – the mean squared distance from each sample to its cluster center.
- …

The quality criterion is chosen often because of its good properties:

- **Convexity:** so that there is a unique, best solution
- **Closed form** for the optimum (linear regression) or at least for the gradient (for SGD).
- An algorithm that **provably converges**.

Model Quality

There are typically other criteria used to measure the quality of models. e.g. for clustering models:

- Silhouette score
- Inter-cluster similarity (e.g. mutual information)
- Intra-cluster entropy

For regression models:

- Stability of the model (sensitivity to small changes)
- Compactness (sparseness or many zero coefficients)

Regularization with Secondary Criteria

While secondary criteria can be measured after the model is built, its too late then to affect the model.

Using secondary criteria **during** the optimization process is called **"regularization".**

Examples:

- **L1 regularization** adds a term to the measure being optimized which is the **sum of absolute value of model coefficients**.
- **L2 regularization** adds a term to the measure being optimized which is the **sum of squares of model coefficients**.

Regularization with Secondary Criteria

L1 regularization in particular is very widely used. It has the following impacts:

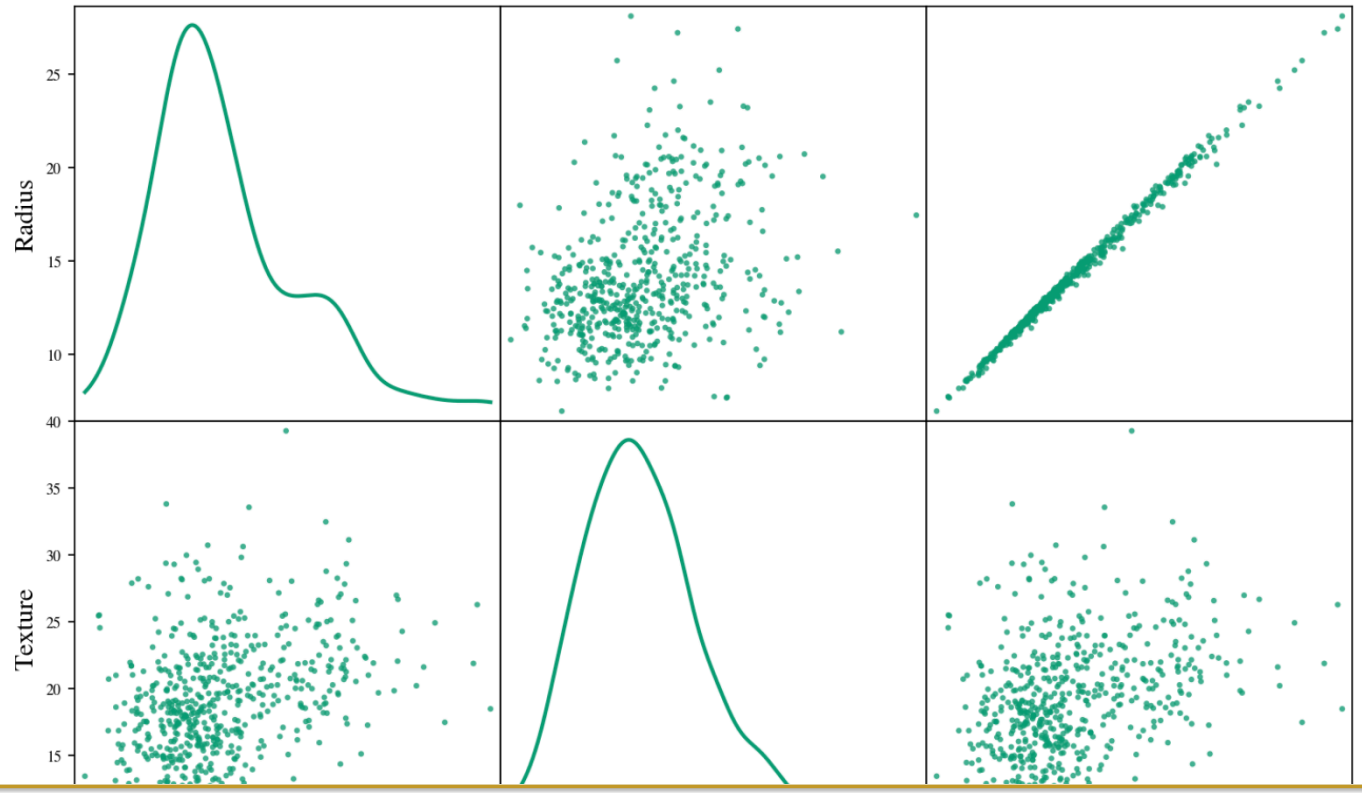
- Yields a **convex optimization** problem in many cases, so there is a unique solution.
- The solution is usually **stable** to small input changes.
- The solution is **quite sparse** (many zero coefficients) and requires less disk and memory to run.
- L1 regularization on factorization models tends to **decrease the correlation** between model factors.

Outline

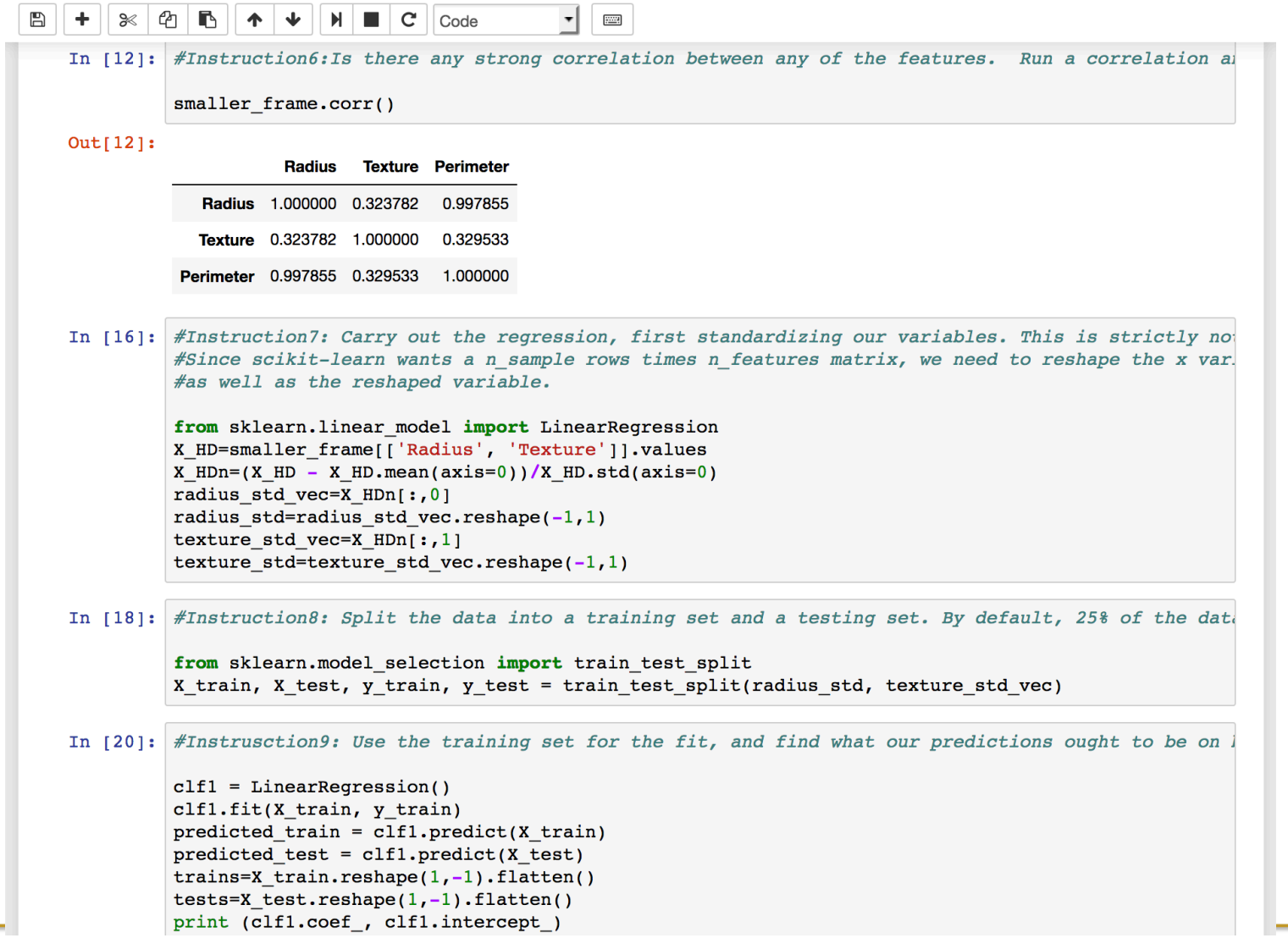
- Supervised Learning
  - Linear Regression
    - R-squared
  - Logistic Regression
  - SVMs
- Gradient-based optimization
- Decision Trees and Random Forests
- **HW 2 Examples**

SPLOM

In [9]:
```
#Instruction5: use a SPLOM to visualize some columns of this dataset. In Panda's the SPLOM is a

smaller_frame=df[['Radius', 'Texture', 'Perimeter']]
from pandas.plotting import scatter_matrix
axeslist=scatter_matrix(smaller_frame, alpha=0.8, figsize=(12, 12), diagonal="kde")
for ax in axeslist.flatten():
    ax.grid(False)
```



Linear Regression



In [12]:
```
#Instruction6:Is there any strong correlation between any of the features.  Run a correlation a

smaller_frame.corr()
```

Out[12]:

	Radius	Texture	Perimeter
Radius	1.000000	0.323782	0.997855
Texture	0.323782	1.000000	0.329533
Perimeter	0.997855	0.329533	1.000000

In [16]:
```
#Instruction7: Carry out the regression, first standardizing our variables. This is strictly no
#Since scikit-learn wants a n_sample rows times n_features matrix, we need to reshape the x var
#as well as the reshaped variable.

from sklearn.linear_model import LinearRegression
X_HD=smaller_frame[['Radius', 'Texture']].values
X_HDn=(X_HD - X_HD.mean(axis=0))/X_HD.std(axis=0)
radius_std_vec=X_HDn[:,0]
radius_std=radius_std_vec.reshape(-1,1)
texture_std_vec=X_HDn[:,1]
texture_std=texture_std_vec.reshape(-1,1)
```

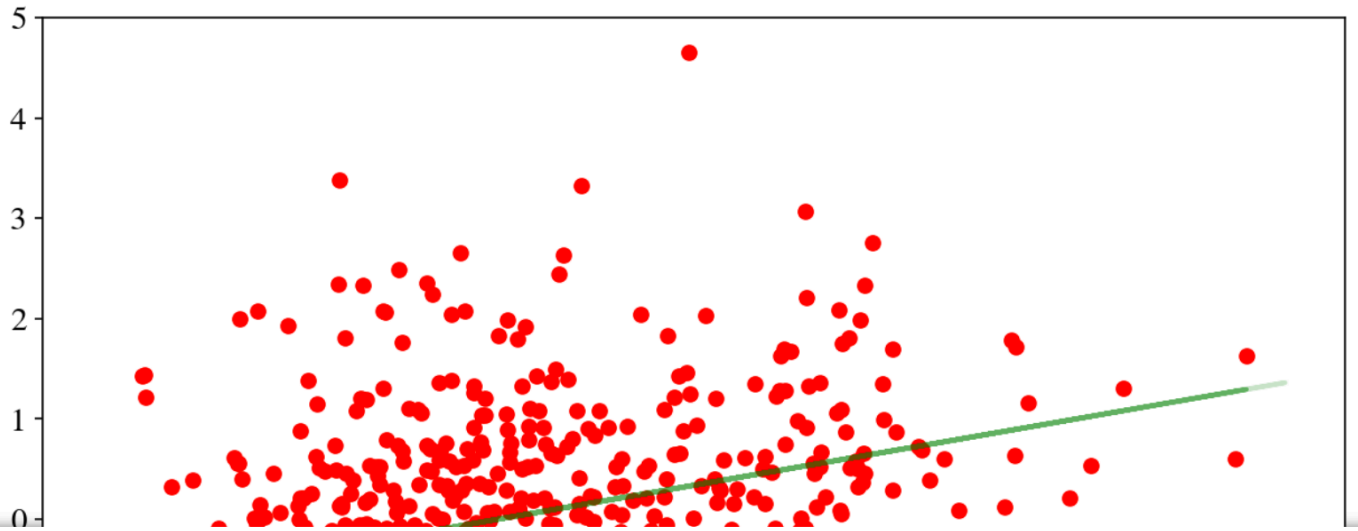
In [18]:
```
#Instruction8: Split the data into a training set and a testing set. By default, 25% of the dat

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(radius_std, texture_std_vec)
```

In [20]:
```
#Instrusction9: Use the training set for the fit, and find what our predictions ought to be on

clf1 = LinearRegression()
clf1.fit(X_train, y_train)
predicted_train = clf1.predict(X_train)
predicted_test = clf1.predict(X_test)
trains=X_train.reshape(1,-1).flatten()
tests=X_test.reshape(1,-1).flatten()
print (clf1.coef_, clf1.intercept_)
```

Linear Regression

In [21]:
```
#Instruction10: Plot the scatter against the fit for both training and test data.

plt.scatter(educ_coll_std_vec, average_income_std_vec,c='r')
plt.plot(trains, predicted_train, c='g', alpha=0.5)
plt.plot(tests, predicted_test, c='g', alpha=0.2)
```

Out[21]:
```
[<matplotlib.lines.Line2D at 0x11a5e3828>]
```



In [23]:
```
#Instruction11: Look at the residuals, again on both sets.

plt.scatter(predicted_test, predicted_test- y_test, c='g', s=40)
plt.scatter(predicted_train, predicted_train- y_train, c='b', s=40, alpha=0.5)
plt.plot([0.4,2],[0,0])
```

Out[23]:
```
[<matplotlib.lines.Line2D at 0x11ab80fd0>]
```

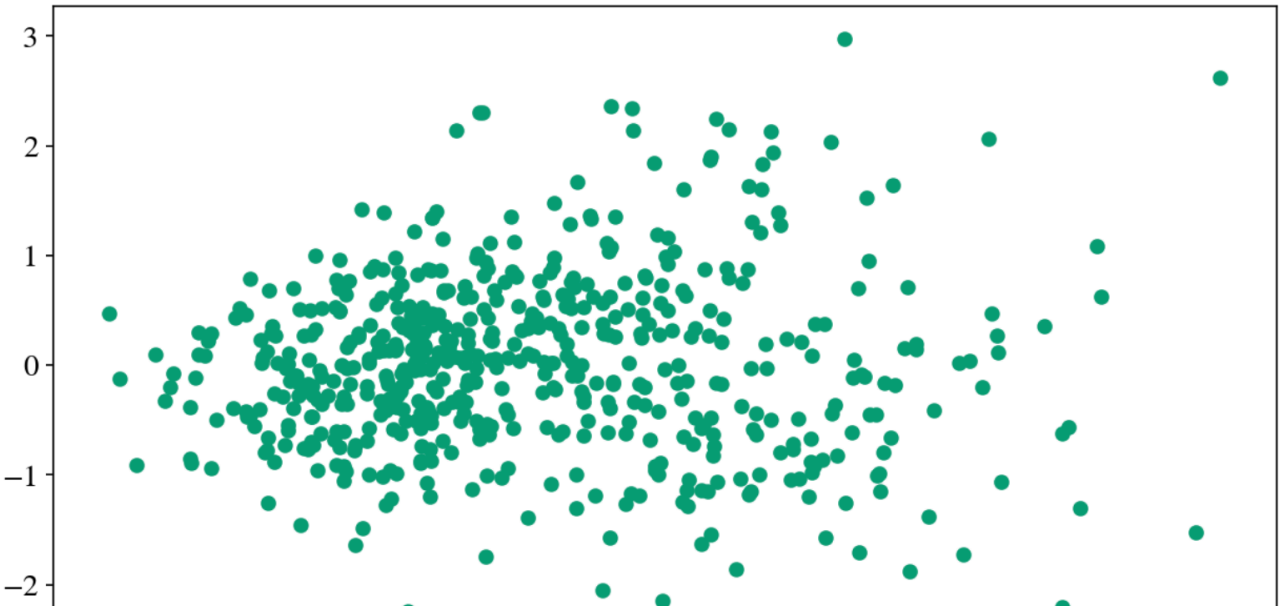
PCA

In [24]:
```
#Instruction12: Take the standarddized data and do a 2-D PCA on it.
#Here we do not seek to accomplish a dimensional reduction, but to understand the variance stru

from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X = pca.fit_transform(X_HDn)
print (pca.explained_variance_ratio_)
```

```
[ 0.66189095  0.33810905]
```

In [25]:
```
plt.scatter(X[:, 0], X[:, 1])
```

Out[25]:
```
<matplotlib.collections.PathCollection at 0x11b467c88>
```

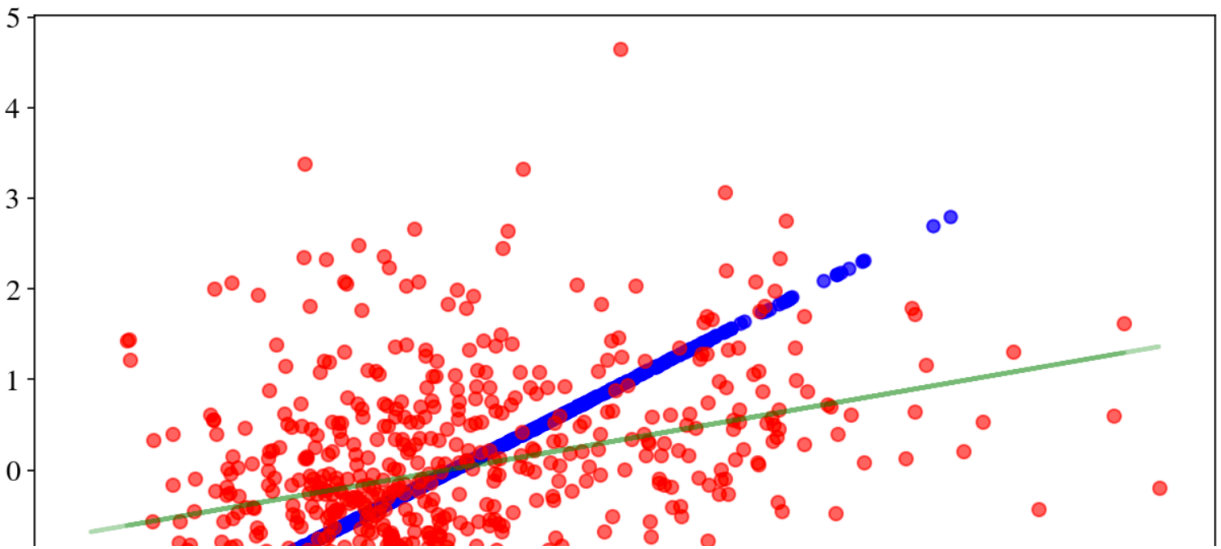


PCA

In [26]:
```
#Instruction13: Reconstruct the original data from only the first component, setting the others

pca1 = PCA(n_components=1) # only keep one dimension!
X_E = pca1.fit_transform(X_HDn)
X_reconstructed = pca1.inverse_transform(X_E)
```

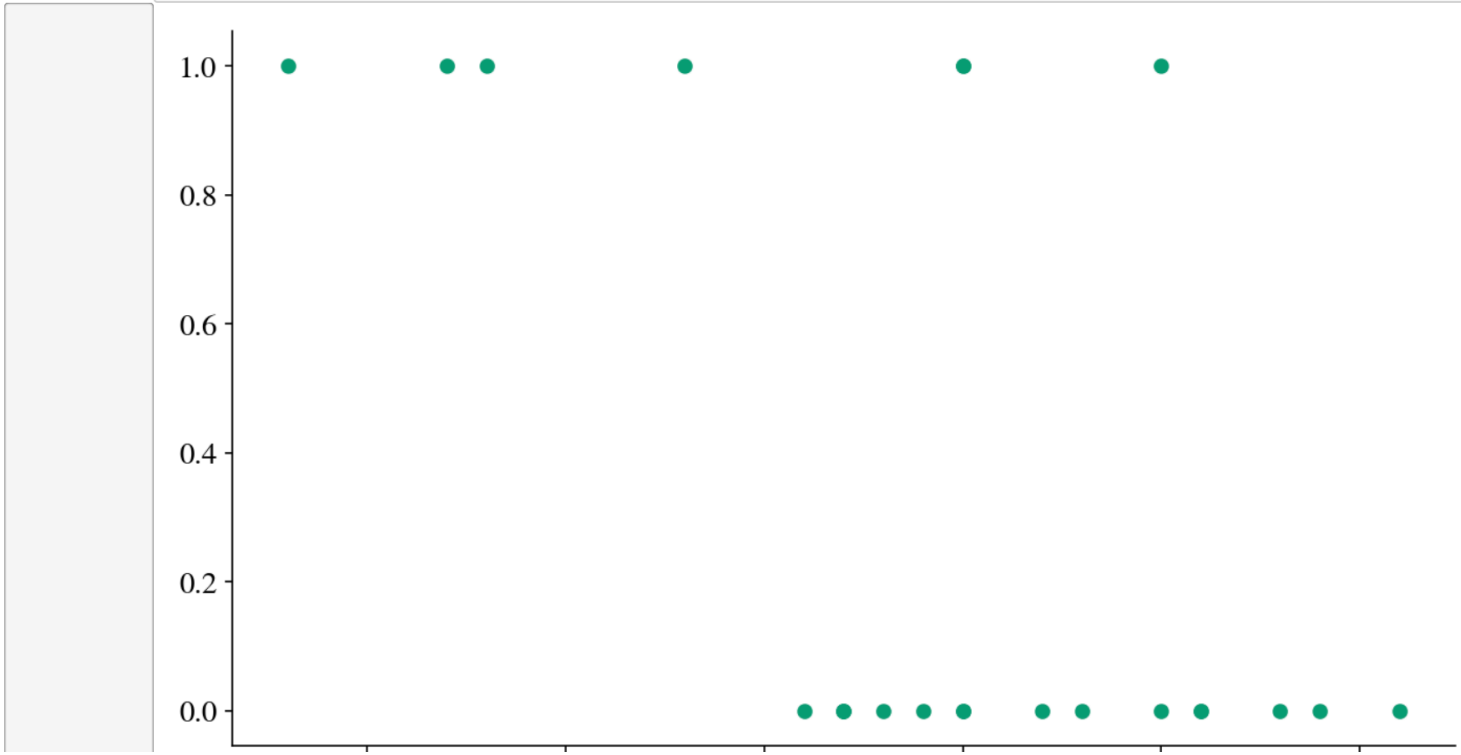
In [27]:
```
#Instruction14: Plot the reconstructed education(x) and income(y) from the first principal comp

plt.scatter(X_reconstructed[:,0], X_reconstructed[:,1],c='b', s=35, alpha=0.7)
plt.scatter(educ_coll_std_vec, average_income_std_vec, s=40, c='r', alpha=0.6)
plt.plot(trains, predicted_train, c='g', alpha=0.3)
plt.plot(tests, predicted_test, c='g', alpha=0.3)
```

Out[27]:
```
[<matplotlib.lines.Line2D at 0x1136eb2b0>]
```



Plotting

```
In [30]: #Instruction2: Plot the array.

         temps, pfail = data[:,0], data[:,1]
         plt.scatter(temps, pfail)
         axes=plt.gca()
         axes.grid(False)
         remove_border(axes)
```



Logistic Regression

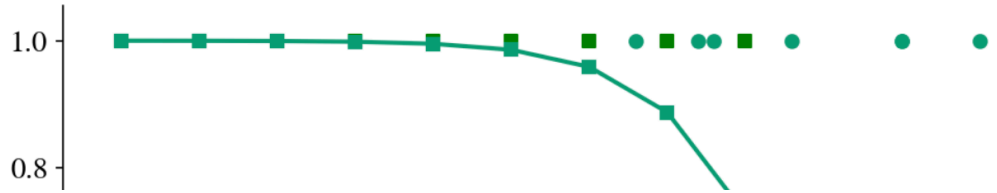
In [31]:
```
#Instruction3: Run Logistic Regression with sklearn.linear_model and set c to 1000.

from sklearn.linear_model import LogisticRegression
reg=1000.
clf4 = LogisticRegression(C=reg)
clf4.fit(temps.reshape(-1,1), pfail)
```

Out[31]:
```
LogisticRegression(C=1000.0, class_weight=None, dual=False,
          fit_intercept=True, intercept_scaling=1, max_iter=100,
          multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,
          solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

In [32]:
```
#Instruction4: Make predictions, get the associated probabilities, and plot them.

tempsnew=np.linspace(20., 90., 15)
probs = clf4.predict_proba(tempsnew.reshape(-1,1))[:, 1]
predicts = clf4.predict(tempsnew.reshape(-1,1))
```

In [33]:
```
plt.scatter(temps, pfail)
axes=plt.gca()
axes.grid(False)
remove_border(axes)
plt.plot(tempsnew, probs, marker='s')
plt.scatter(tempsnew, predicts, marker='s', color="green")
```

Out[33]:
```
<matplotlib.collections.PathCollection at 0x1109464e0>
```



Validation

In [34]:
```
#Instruction5: use pandas crosstab to write a table of prediction vs failure on the "training" set.

pd.crosstab(pfail, clf4.predict(temps.reshape(-1,1)), rownames=["Actual"], colnames=["Predicted"])
```

Out[34]:

Predicted	0.0	1.0
Actual		
0.0	16	0
1.0	3	4