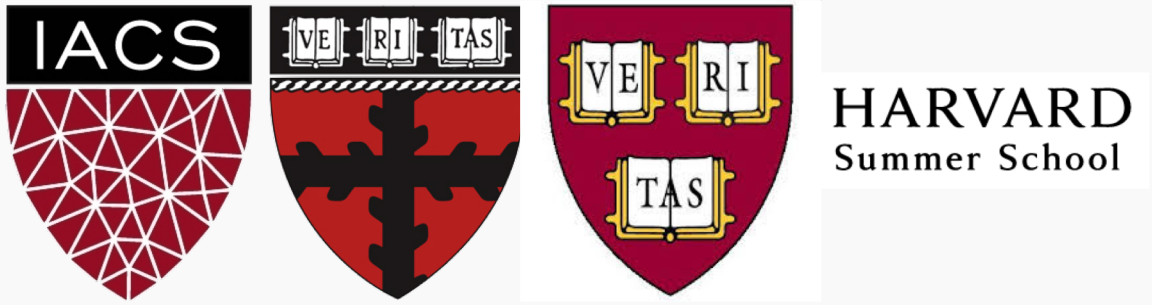
Lecture #8: Decision Trees and Random Forests

CS-S109A: Introduction to Data Science

Kevin Rader

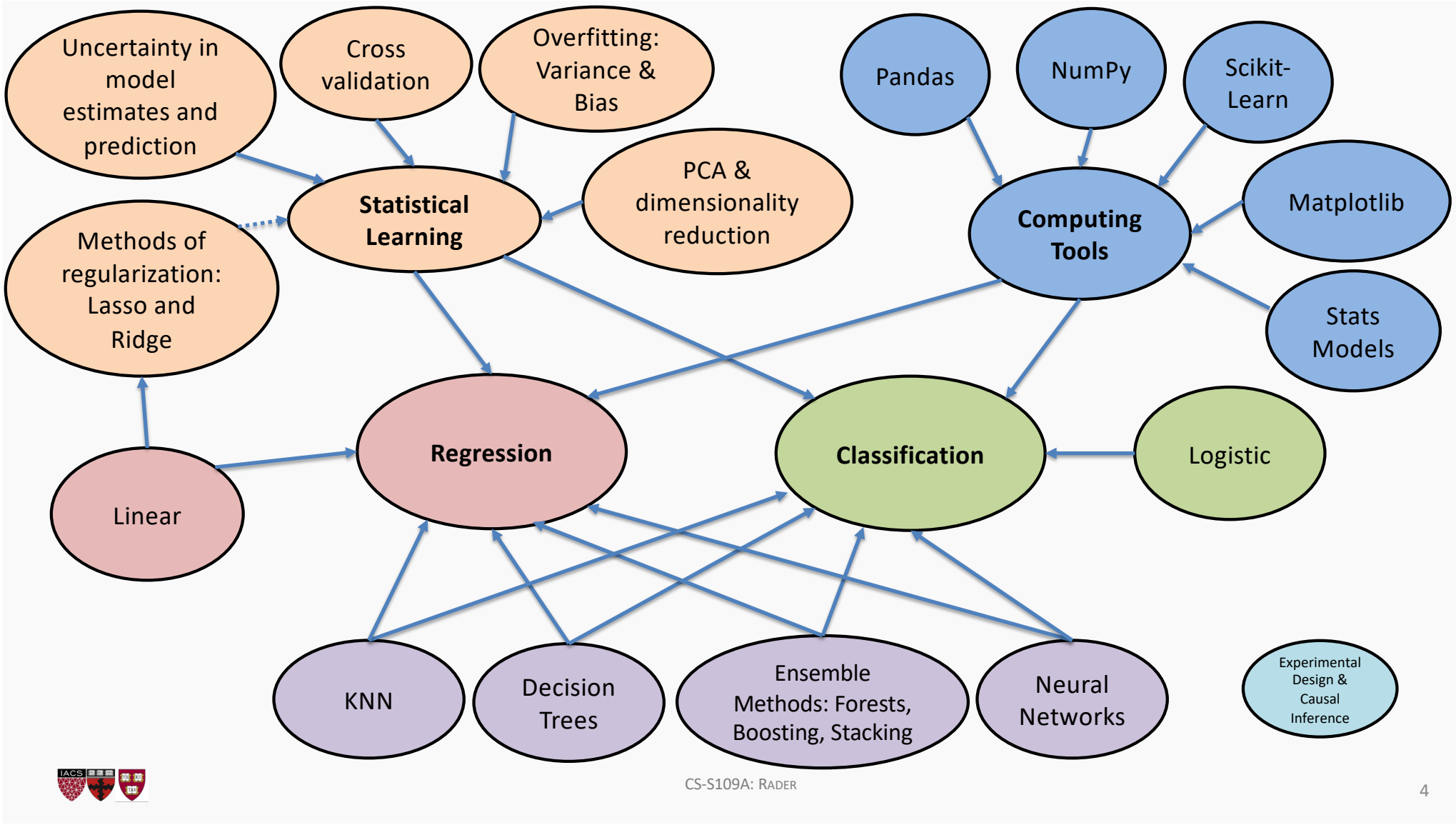


ANNOUNCEMENTS

- **HW3** grades will be released tomorrow.
- **HW4** is due tomorrow night at 11:59pm
  - Please see the **Ed Announcement** regarding errata and clarifications.
- **HW5** (our last HW) will be posted tomorrow night, and due 7/28. It will be shorter.
- The **Final Exam** (individual work) will be posted July 27 (Monday) and due Monday, Aug. 3. It is cumulative and will be more open-ended.

Outline

- General Review
- Review of Ensemble Methods
- Boosting
  - Set-up and intuition
  - Connection to Gradient Descent
  - The Algorithm
- Stacking


Uncertainty in model estimates and prediction
Cross validation
Overfitting: Variance & Bias
Pandas
NumPy
Scikit-Learn
Statistical Learning
PCA & dimensionality reduction
Computing Tools
Matplotlib
Methods of regularization: Lasso and Ridge
Stats Models
Regression
Classification
Logistic
Linear
KNN
Decision Trees
Ensemble Methods: Forests, Boosting, Stacking
Neural Networks
Experimental Design & Causal Inference

Lectures 1 & 2: Data

We started the class with basic data analysis and got experience with the tools within Python to do data acquisition, and EDA:

- Tools: numpy, pandas, matplotlib, beautifulsoup

We learned basic **exploratory data analysis** (EDA) approaches:

- Summaries: basic statistics, tables, group comparisons
- Visuals: histograms, boxplots, barplots; scatterplots, side-by-side boxplots, stacked and side-by-side barplots

The keys to decide what summaries to look at were 2 fold:

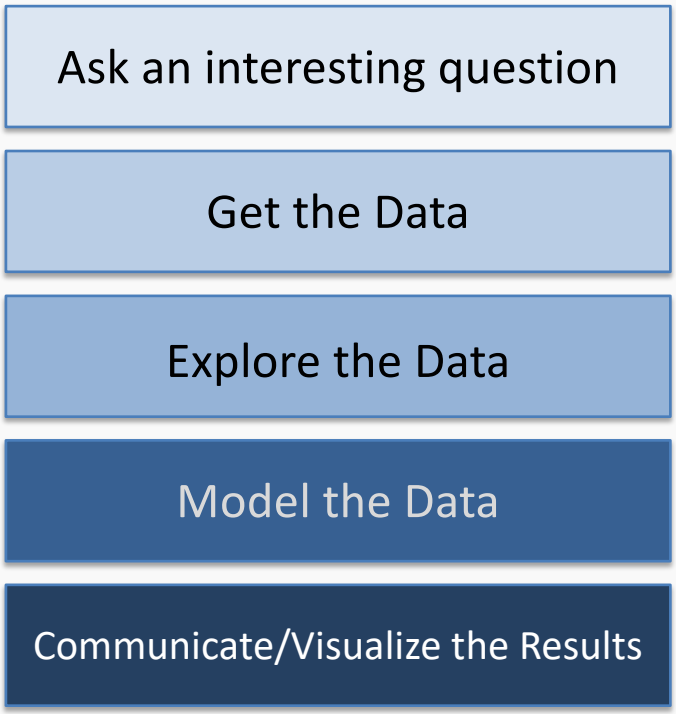
1. What type of variable is the response variable (quantitative vs. categorical)?

2. What comparisons and associations do we want to inspect? Or just explore the distribution.

We also first saw the data science process…

What?

The Data Science Process

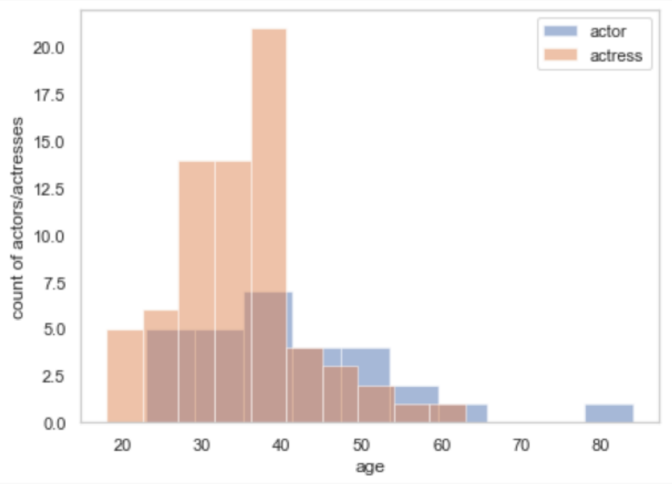


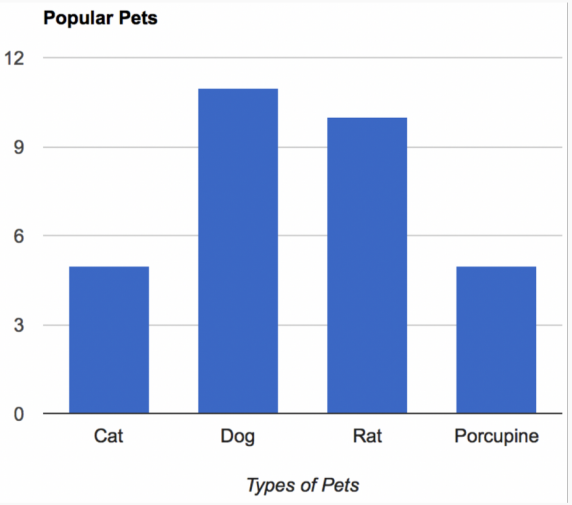
Lectures 1 & 2: Concept Checks

When is the median preferred to be used over the mean? Why?

What is an outlier? How can they be detected?

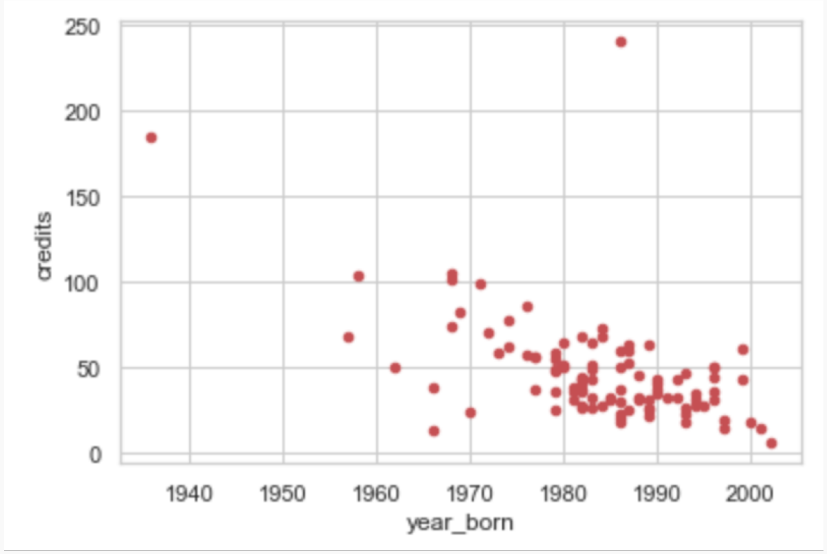
Describe these distributions:

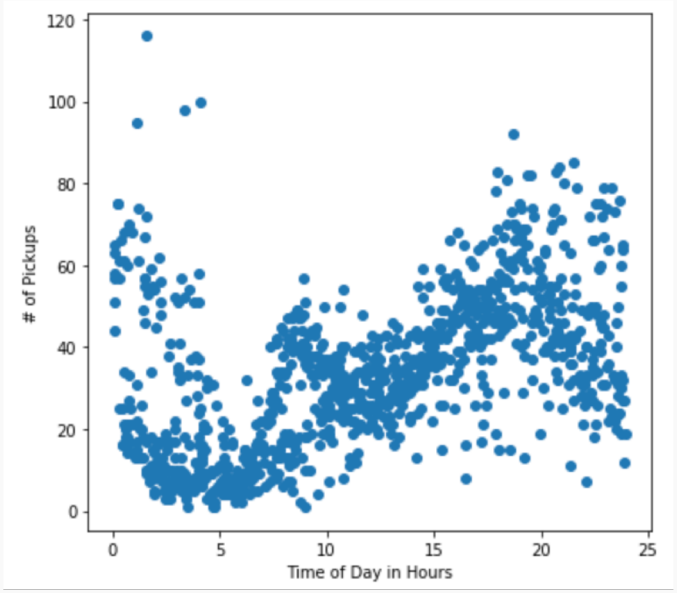




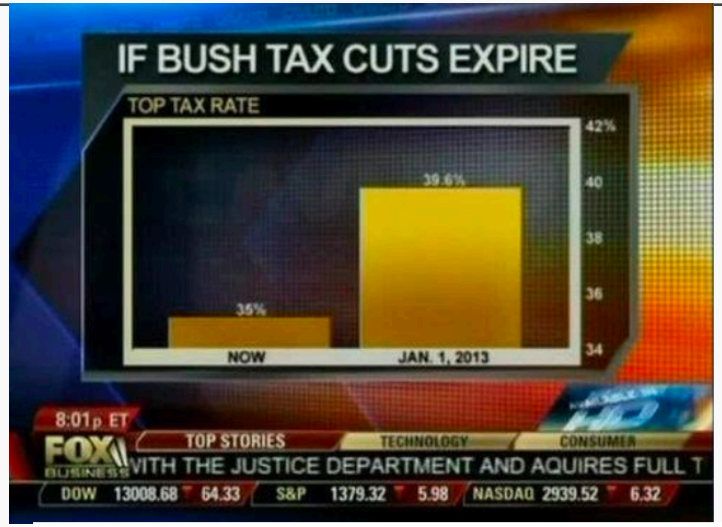
Lectures 1 & 2: Concept Checks (cont.)

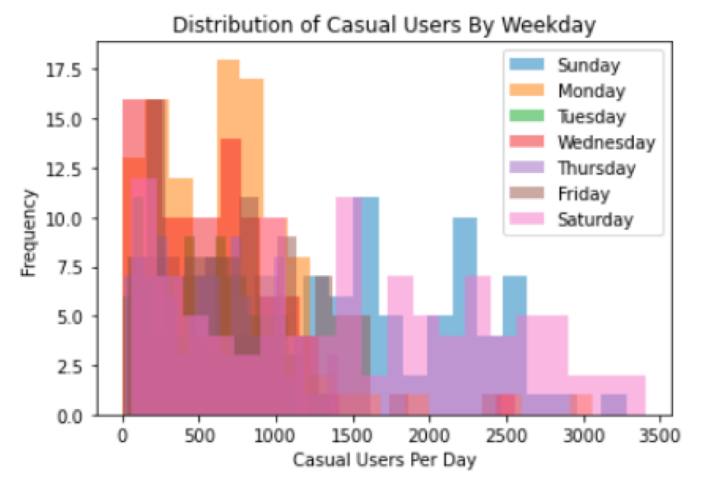
Describe these relationships:

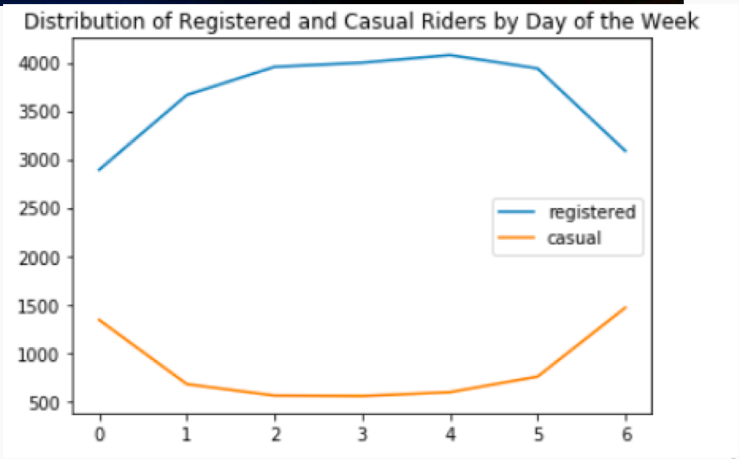


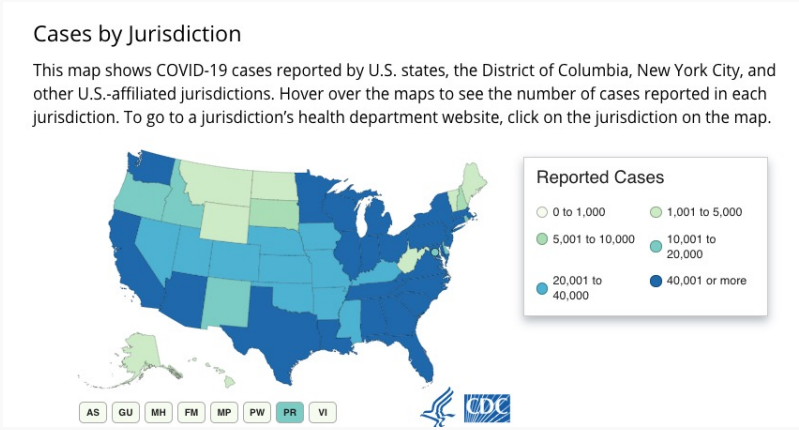


Lectures 1 & 2: Concept Check: Adventures in Plotting









Lecture 3: *k*-NN and Linear Regression (cont.)

We first started doing **statistical modeling :**

$$Y = f(\boldsymbol{X}) + \varepsilon$$

We saw our first 2 regression-type models:

1. *k* Nearest Neighbors (*k*-NN): $\hat{y}_i = \frac{1}{k}\sum_{j=1}^{k} y_{n_j}$
2. Linear Regression: $\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$

Mean Squared Error (MSE) is the standard optimization function that these models use:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

The estimates for linear regression have a closed form solution:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

Lecture 3: *k*-NN and Linear Regression

There is a probabilistic basis of linear regression (led to 4 assumptions):

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i, \text{ where } \varepsilon_i \sim N(0, \sigma^2)$$

This led to our estimates having **sampling distributions**. For example:

$$\hat{\beta}_1 \sim N\left(\beta_1, \frac{\sigma^2}{\sum(x_i - \bar{x})^2}\right)$$

We first learned how to **bootstrap** data to build sampling distributions (and perform inferences) when probabilistic assumptions are violated.

We first saw the difference between **inference** (and interpretations) and **prediction** problems.

For a prediction application: We first learned about using a **test set** to compare the accuracy of models (as measured by MSE or R^2 for regression problems).

Lecture 3: Concept Checks

Which is a parametric model? Which is a non-parametric model? Why? What does it mean to be non-parametric vs. parametric?

How are neighbors chosen in *k*-NN? How are predictions made? How can this model be interpreted? What happens when $k = 1$? When $k = n$?

Why is it called *ordinary least squares* (OLS) regression?

What happens when the assumptions to linear regression are violated?

How can bootstrapping be used to calculate confidence intervals?

Lecture 4: Multiple Regression

Multiple regression uses many predictors to model the response variable parametrically:

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_1 x_{i,2} \ldots + \beta_1 x_{i,p}$$

We formulated the multiple regression problem as a linear algebra problem:

$$\vec{Y} = \mathbf{X}\vec{\beta} + \vec{\varepsilon}$$

Which gave us closed form solutions to estimate the parameters:

$$\hat{\vec{\beta}} = \left(\mathbf{X}^\mathbf{T}\mathbf{X}\right)^{-1}\mathbf{X}^\mathbf{T}\vec{Y}, \qquad \hat{\sigma}^2 = \frac{1}{n-p-1}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

We saw the *F*-test to determine if a model is worthwhile overall.

We learned the difference between a confidence interval and a prediction interval.

We first witnessed multicollinearity.

Lecture 4: Multiple Regression (cont.)

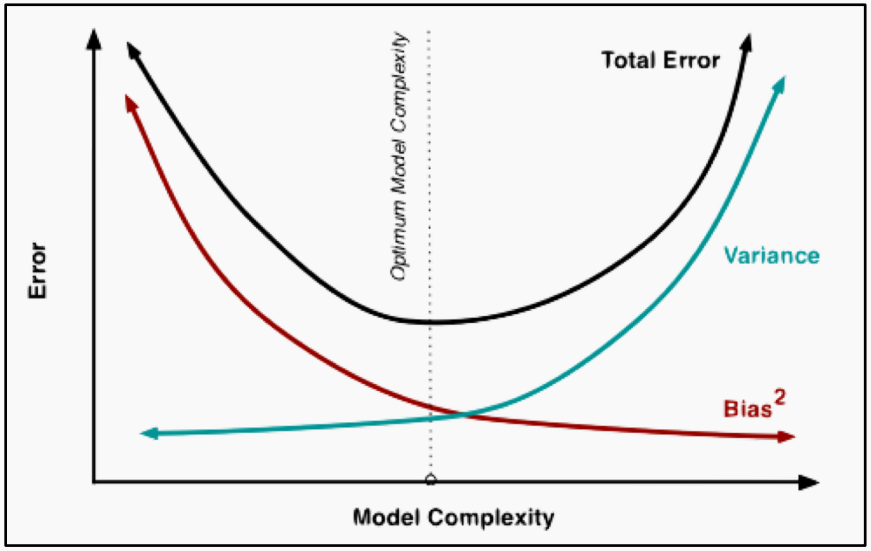
We parameterized the models with various types of predictors:

- binary indicators
- polynomial functions
- interaction terms

We observed the models with lots of predictors (complex models) lead to **overfitting**.

We first learned the variance-bias trade-off when fitting models.

We learned one way to prevent overfitting is by using variable selection.



Lecture 4: Concept Checks

What is the interpretation of $\hat{\beta}_1$ in a multiple regression model?

What happens when multicollinearity is present? Why is multicollinearity bad? Why is it not always bad? How can this be handled?

What is the interpretation of a confidence interval for $\hat{\beta}_1$ in a multiple regression model?

What is polynomial regression used for?

How does a multiple regression model for 2 predictors (one binary, the other quantitative) compare in interpretation when the interaction is included vs. is not included in the model. How can we visualize this?

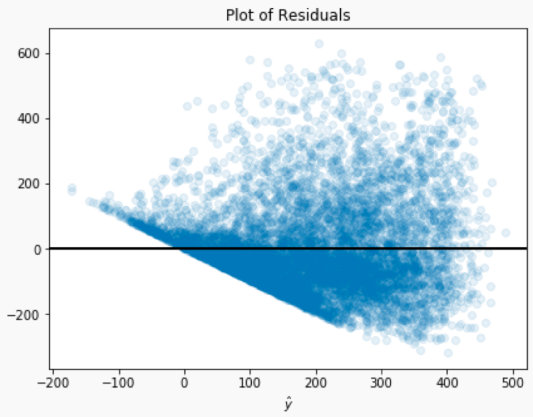
Why is one of the binary terms dropped out when modeling a categorical predictors? What are the interpretations of the $\hat{\beta}_j$ in this model?

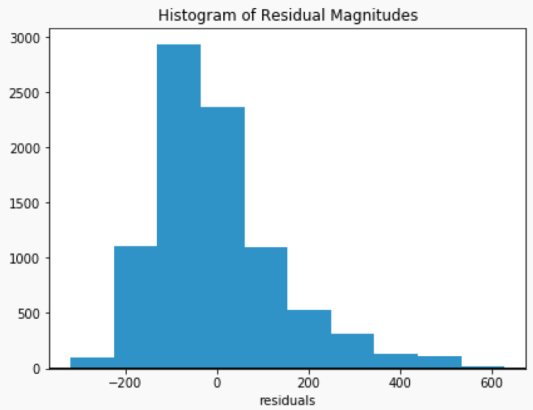
Lecture 4: Concept Checks (cont.)

When is an *F*-test useful?

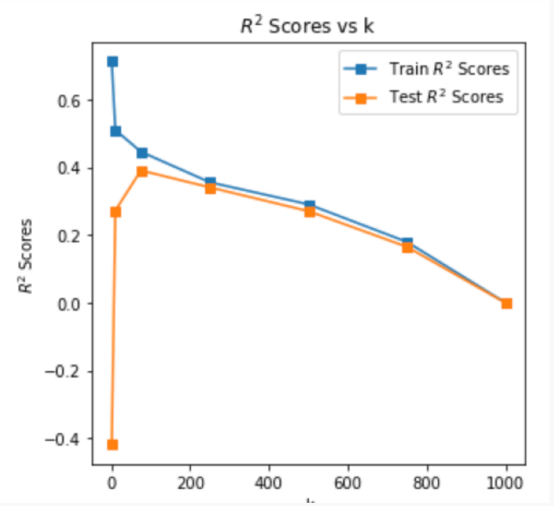
What should be used as a metric when comparing models (in variable selection, for example)?

Interpret these residual plots:





What does this plot show?



Lecture 5: Cross-Validation and Regularization

We learned the models should be compared over many validation sets in order to overfit to the test set: aka **cross validation**.

We saw *k*-fold and random subsets cross validation.

We saw the use of train-validation-test splits!

Regularized regression penalizes MSE in order to reduce the affect of overfitting. Specifically:

LASSO: $$L_{LASSO}(\beta) = \frac{1}{n}\sum_{i=1}^{n}|y_i - \boldsymbol{\beta}^\top \boldsymbol{x}_i|^2 + \lambda \sum_{j=1}^{J}|\beta_j|.$$

Ridge: $$L_{Ridge}(\beta) = \frac{1}{n}\sum_{i=1}^{n}|y_i - \boldsymbol{\beta}^\top \boldsymbol{x}_i|^2 + \lambda \sum_{j=1}^{J}\beta_j^2.$$

The penalization term (λ) should be tuned to minimize out-of-sample error.

Lecture 5: Concept Checks

What are the advantages to *k*-fold cross validation vs. random subsets cross-validation?

Why is the intercept term not penalized in regularized methods? What if it was penalized?

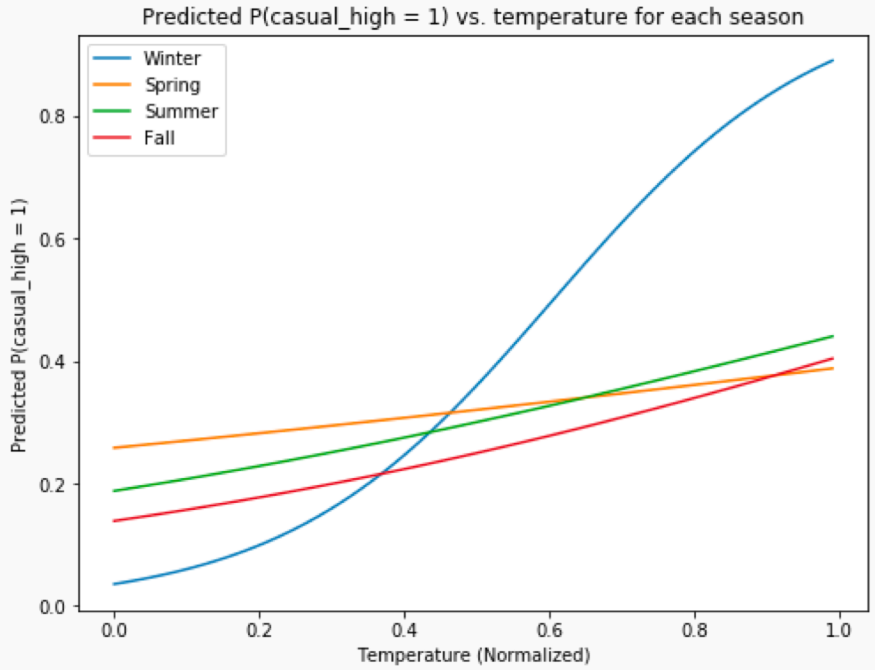
When tuning λ, what loss function should be used?

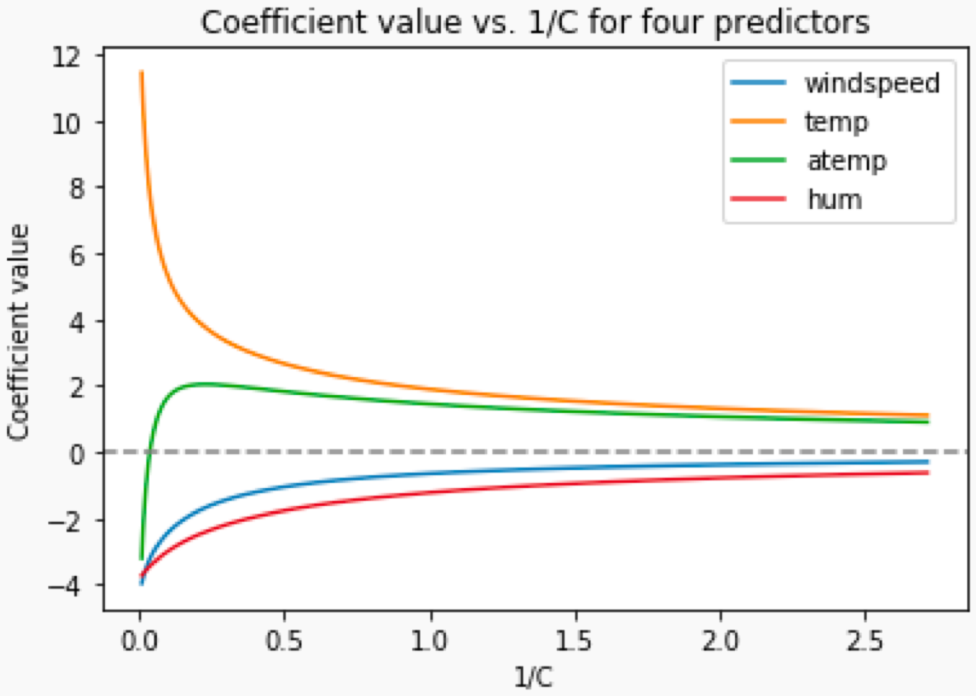
When is LASSO preferred over Ridge? What about the other way around?

How do regularization methods affect multicollinearity?

Lecture 5: Concept Checks (cont.)

What is interesting about these plots? What do they illustrate?





Lecture 6: Classification and Logistic Regression

Logistic regression models the log-odds of success in a binary outcome variable:

$$\ln\left(\frac{P(Y=1)}{1-P(Y=1)}\right) = \beta_0 + \beta_1 X.$$

The coefficients are estimated by minimizing negative log-0likelihood for the generalized linear model:

$$loss(p|Y) = -\sum_i \left[y_i \log \frac{1}{1+e^{-\beta X_i}} + (1-y_i)\log\left(1-\frac{1}{1+e^{-\beta X_i}}\right)\right]_i$$

All of the machinery of multiple linear regression (use of various predictors, interpretation adjustment, penalizationetc.) apply to multiple logistic regression.

Logistic regression can be generalized to the multi-class problem via multinomial logistic regression or one-vs.-rest (OVR) logistic regression (the default in sklearn)

Lecture 6: Classification and Logistic Regression (cont.)

A logistic regression's predictions are typically turned into classifications by choosing the Bayes' classifier to minimize misclassification error rate. This leads to a linear classification boundary.

Confusion Matrices can be used for many classification thresholds, and positive predictive values (PPV) and NPV rates can be calculated from them.

An ROC curve depicts all of these combinations of NPV and PPV, and can be summarized via the area under the curve metric (AUC).

There are many metrics to use for determining accuracy of classification models (AUC, PPV, NPV, for examples).

Lecture 6: Concept Checks

What is the interpretation of $\hat{\beta}_1$ in a multiple logistic regression problem?

How is the loss function minimized in logistic regression?

What happens in logistic regression when there is *perfect separation*?

Why are the logistic regression classification boundaries linear? How can this be adapted to be non-linear?

What is the loss function that is penalized in regularized multiple logistic regression?

Lecture 7: *k*-NN Classification and PCA

The method of *k*-NN can be adapted to a classification problem: the nearest neighbors are calculated the same way as a regression problem, but the predictions are just relative frequency of classes within the neighborhood.

k-NN classification boundaries are very flexible, but need to be tuned to prevent overfitting.

High Dimensionality (a lot of predictors) leads to numerical and estimation issues (unidentifiability, difficulty in interpretation, and overfitting.

Principal Components Analysis (PCA) is a method to handle high dimensionality in the predictors. It is an unsupervised method.

In PCA, a set of new predictors (the components) is created that is an ordered list of representative variables based on the explained variability in the predictor set.

It is VERY important to standardize your predictors before performing PCA.

Lecture 7: Concept Checks

Should predictors be standardized in *k*-NN? How can categorical predictors be handled?

How can probability predictions be made in *k*-NN classification?

When is PCA used most commonly in practice?

Is the first PCA component going to be the best predictor in a PCR regression? Why?

How many PCA components should be used in a PCR egression?

How can the coefficient estimates be interpreted in a regression fit on PCA component vectors?

Lecture 8: Trees and Forests

Decision trees (both regression and classification) are calculated by finding the splits in the predictor set (one at a time) that best improve some optimization metric (MSE for regression, Gini or Entropy for classification). This leads to a step-like function in the prediction scatterplots or classification boundaries.

Overfitting can be prevented by either some stopping criterion or by pruning an overfit tree

Ensemble methods (combining the predictions of many base models) can be used to improve prediction accuracy of models. This makes the interpretation difficult.

Bootstrap-Aggregating (Bagging) is the first ensemble method we have seen. It improves the prediction of by averaging the predictions of many base tree models. Each base model is estimated based on a random bootstrapped subset of observations.

Lecture 8: Trees and Forests (cont.)

Random Forests adds an extra layer of random sampling (a subset of the variables at each split) to de-correlate the base tree models.

There are 3 parameters to tune in a random forest: complexity (depth measured in some way), number of predictors sampled at each split, and number of trees.

Out-of-bag sample can be used to estimate out-of-sample error, and thus can be used to tune any hyperparameters.

Feature importance can be measured to help interpret a tree-based model. This measures the contributed improvement in MSE (or Gini/Entropy)

Lecture 8: Concept Checks

Why should categorical predictors not be turned into binary indicator variables in a tree-based model?

Should predictors be standardized in a tree-based model? Should they be log-transformed? What about the response variable? How are interactions handled?

How can relationship between a predictor with the response be determined in a tree-based model? What values for the other predictors should be used when doing this?

How is bagging a special case of random forests?

Why do bagging and random forests improve the variance of the overly complex based models?

Which parameters really need to be tuned in a random forest?

Review of Ensemble Methods

(Bagging and Random Forests)

Bags and Forests of Trees

Last time we examined how the short-comings of single decision tree models can be overcome by ensemble methods - making one model out of many trees.

We focused on the problem of training large trees, these models have low bias but high variance.

We compensated by training an ensemble of full decision trees and then averaging their predictions - thereby reducing the variance of our final model.

Bags and Forests of Trees (cont.)

Bagging:

- create an ensemble of full trees, each trained on a bootstrap sample of the training set;
- average the predictions

Random forest:

- create an ensemble of full trees, each trained on a bootstrap sample of the training set;
- in each tree and each split, randomly select a subset of predictors, choose a predictor from this subset for splitting;
- average the predictions

Note that the ensemble building aspects of both method are embarrassingly parallel!

Motivation for Boosting

Could we address the shortcomings of single decision trees models in some other way?

For example, rather than performing variance reduction on complex trees, can we decrease the bias of simple trees - make them more expressive?

A solution to this problem, making an expressive model from simple trees, is another class of ensemble methods called ***boosting***.

Boosting Algorithms

Gradient Boosting

The key intuition behind boosting is that one can take an ensemble of simple models $\{T_h\}_{h \in H}$ and additively combine them into a single, more complex model.

Each model T_h might be a poor fit for the data, but a linear combination of the ensemble

$$T = \sum_h \lambda_h T_H$$

can be expressive/flexible.

But which models should we include in our ensemble? What should the coefficients or weights in the linear combination be?

Gradient Boosting: the algorithm

Gradient boosting is a method for iteratively building a complex regression model *T* by adding simple models. Each new simple model added to the ensemble compensates for the weaknesses of the current ensemble.

1. Fit a simple model $T^{(0)}$ on the training data

$$\{(x_1, y_1), \ldots, (x_N, y_N)\}$$

   Set $T \leftarrow T^{(0)}$. Compute the residuals $\{r_1, \ldots, r_N\}$ for T.

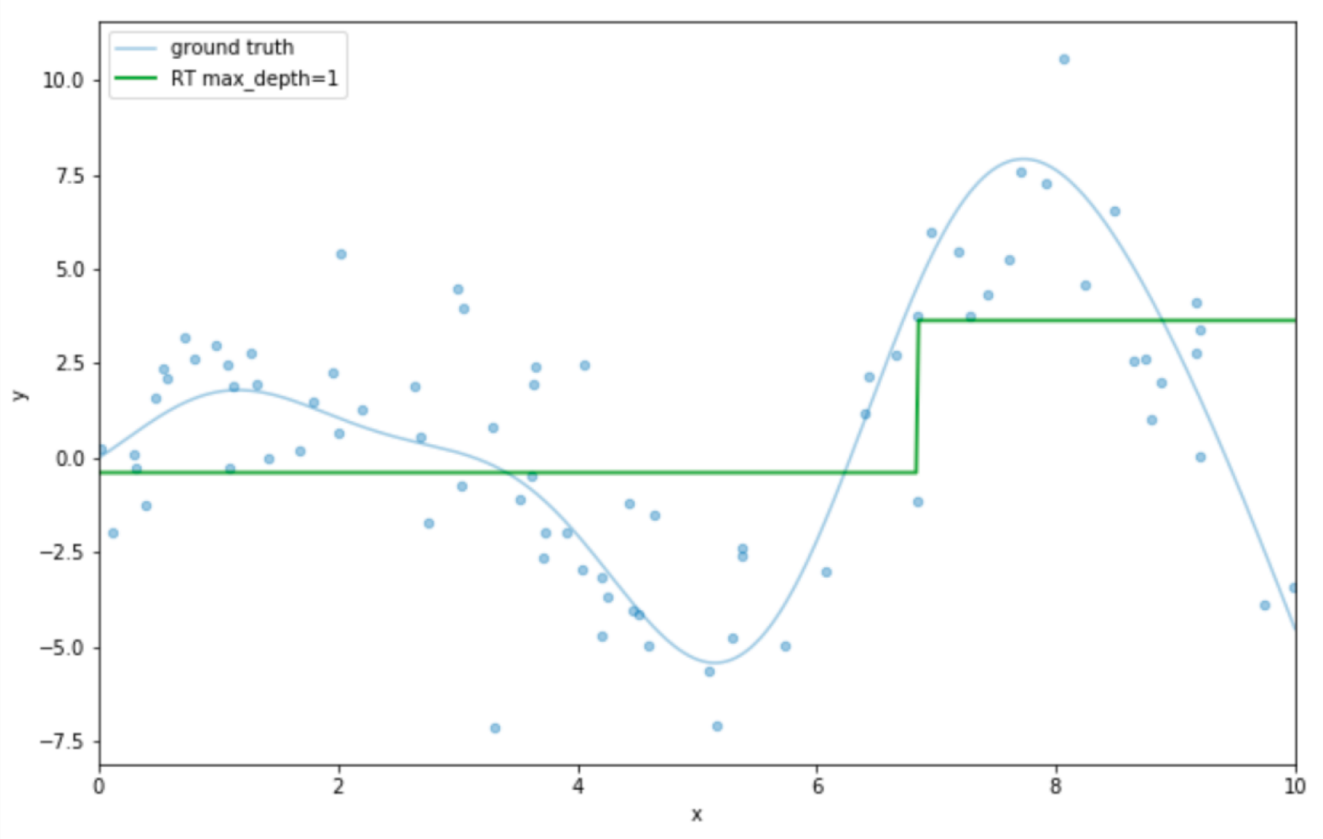
2. Fit a simple model, T^i, to the current ***residuals***, i.e. train using

$$\{(x_1, r_1), \ldots, (x_N, r_N)\}$$

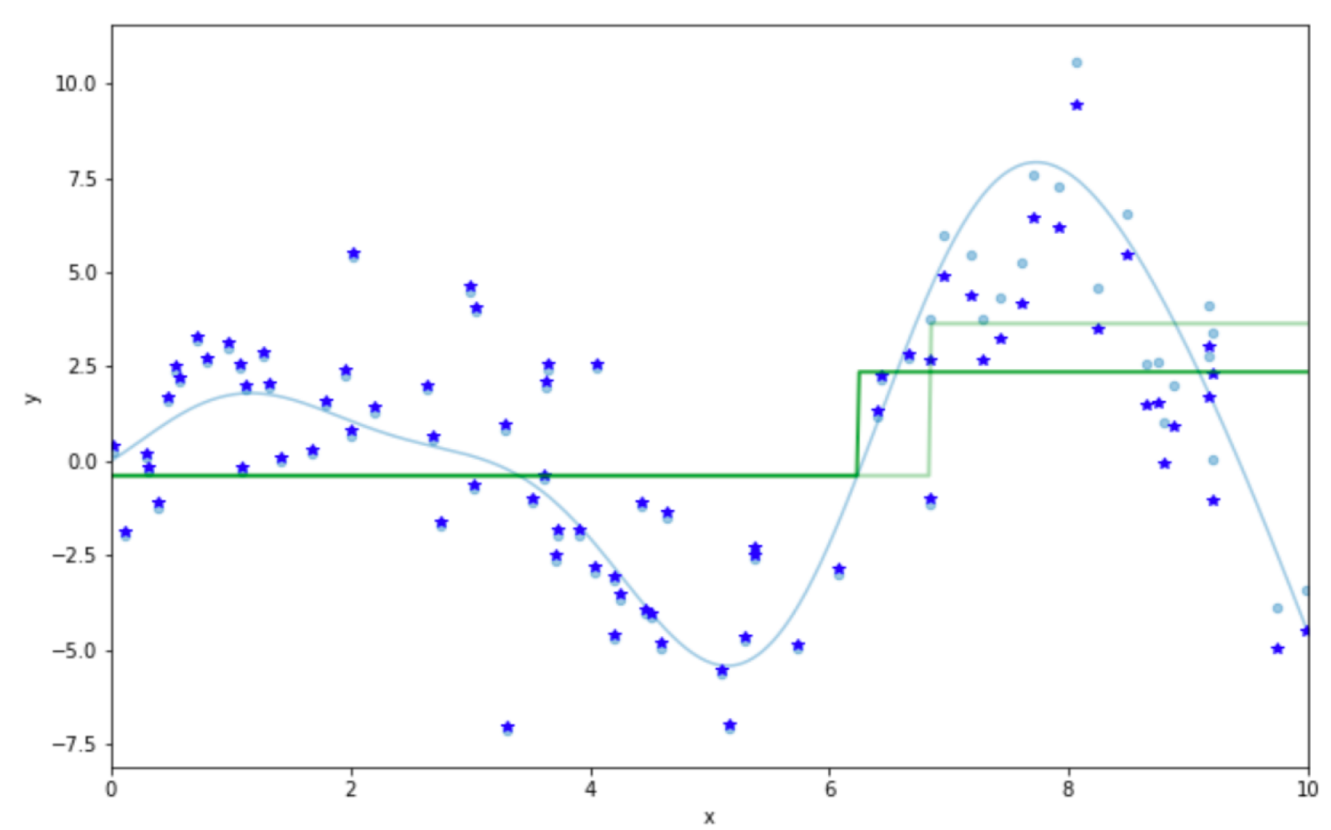
3. Set $T \leftarrow T + \lambda T^i$
4. Compute residuals, set $r_n \leftarrow r_n - \lambda T^i(x_n)$, $n = 1,\ldots,N$
5. Repeat steps 2-4 until ***stopping*** condition met.

where λ is a constant called the ***learning rate***.

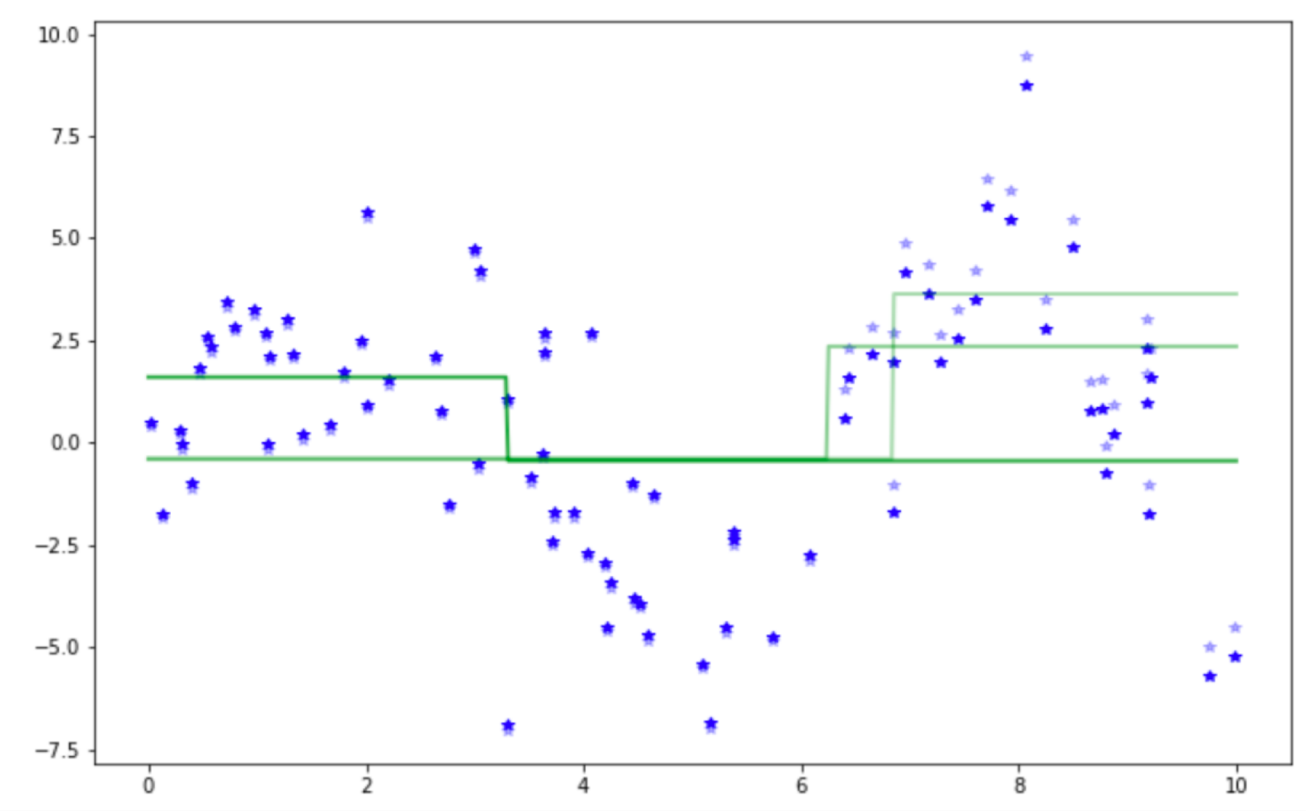
Gradient Boosting: illustration (0)



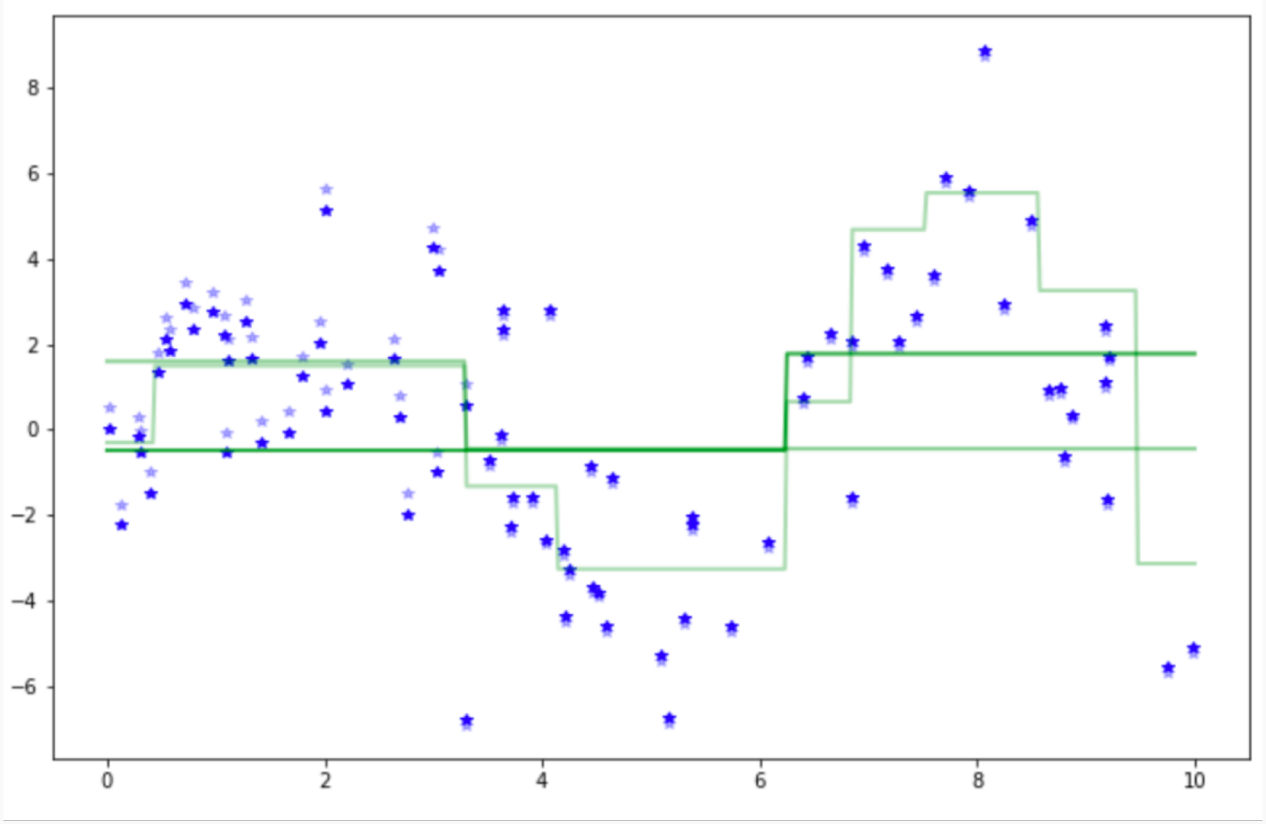
Gradient Boosting: illustration 1



Gradient Boosting: illustration 2



Gradient Boosting: illustration 3



Why Does Gradient Boosting Work?

Intuitively, each simple model $T^{(i)}$ we add to our ensemble model T, models the errors of T.

Thus, with each addition of $T^{(i)}$, the residual is reduced

$$r_n - \lambda T^{(i)}(x_n)$$

Note that gradient boosting has a tuning parameter, λ.

If we want to easily reason about how to choose λ and investigate the effect of λ on the model T, we need a bit more mathematical formalism.

In particular, how can we effectively descend through this optimization via an iterative algorithm?

We need to formulate gradient boosting as a type of ***gradient descent***.

Review: A Brief Sketch of Gradient Descent

In optimization, when we wish to minimize a function, called the ***objective function***, over a set of variables, we compute the partial derivatives of this function with respect to the variables.

If the partial derivatives are sufficiently simple, one can analytically find a common root - i.e. a point at which all the partial derivatives vanish; this is called a ***stationary point.***

If the objective function has the property of being ***convex***, then the stationary point is precisely the min.

Review: A Brief Sketch of Gradient Descent the Algorithm

In practice, our objective functions are complicated and analytically find the stationary point is intractable.

Instead, we use an iterative method called ***gradient descent***:

1. Initialize the variables at any value:

$$x = [x_1, ..., x_J]$$

2. Take the gradient of the objective function at the current variable values:

$$\nabla f(x) = \left[\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_J}(x)\right]$$

3. Adjust the variables values by some negative multiple of the gradient:

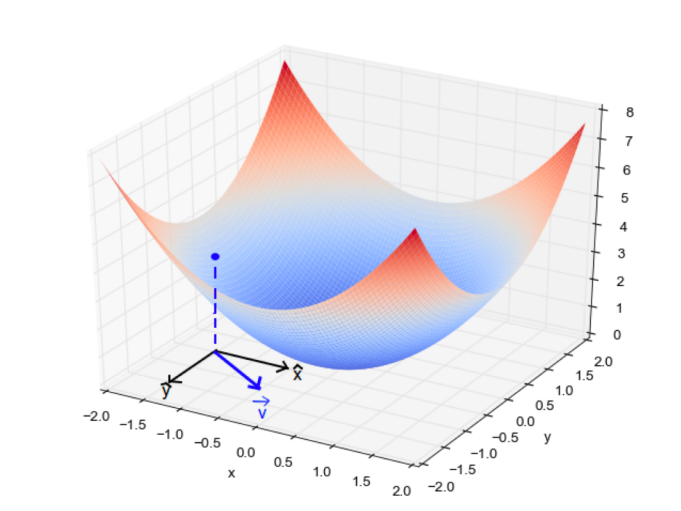
$$x \leftarrow x - \lambda \nabla f(x)$$

The factor λ is often called the learning rate.

Why Does Gradient Descent Work?

Claim: If the function is convex, this iterative methods will eventually move x close enough to the minimum, for an appropriate choice of λ.

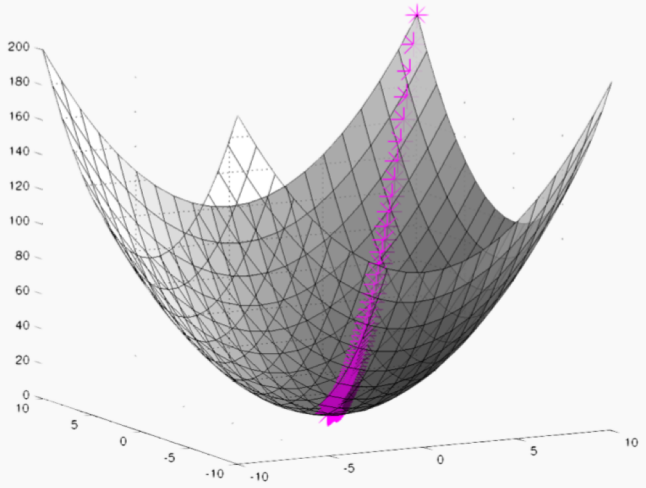
Why does this work? Recall, that as a vector, the gradient at at point gives the direction for the greatest possible rate of increase.



Why Does Gradient Descent Work?

Subtracting a λ multiple of the gradient from x, moves x in the ***opposite*** direction of the gradient (hence towards the steepest decline) by a step of size λ.

If *f* is convex, and we keep taking steps descending on the graph of *f*, we will eventually reach the minimum.



Gradient Boosting as Gradient Descent

Often in regression, our objective is to minimize the MSE

$$\text{MSE}(\hat{y}_1, \ldots, \hat{y}_N) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Treating this as an optimization problem, we can try to directly minimize the MSE with respect to the predictions

$$\begin{aligned} \nabla \text{MSE} &= \left[\frac{\partial \text{MSE}}{\partial \hat{y}_1}, \ldots, \frac{\partial \text{MSE}}{\partial \hat{y}_N} \right] \\ &= -2 \left[y_1 - \hat{y}_1, \ldots, y_N - \hat{y}_N \right] \\ &= -2 \left[r_1, \ldots, r_N \right] \end{aligned}$$

The update step for gradient descent would look like

$$\hat{y}_n \leftarrow \hat{y}_n + \lambda r_n, \quad n = 1, \ldots, N$$

Gradient Boosting as Gradient Descent (cont.)

There are two reasons why minimizing the MSE with respect to $\hat{y}_n$'s is not interesting:

- We know where the minimum MSE occurs: $\hat{y}_n = y_n$, for every n.
- Learning sequences of predictions, $\hat{y}_n^1, \ldots, \hat{y}_n^i, \ldots$, does not produce a model. The predictions in the sequences do not depend on the predictors!

Gradient Boosting as Gradient Descent (cont.)

The solution is to change the update step in gradient descent. Instead of using the gradient - the residuals - we use an ***approximation*** of the gradient that depends on the predictors:

$$\hat{y} \leftarrow \hat{y}_n + \lambda \hat{r}_n(x_n), \quad n = 1, \ldots, N$$

In gradient boosting, we use a simple model to approximate the residuals, $\hat{r}_n(x_n)$, in each iteration.

Motto: gradient boosting is a form of gradient descent with the MSE as the objective function.

Technical note: note that gradient boosting is descending in a space of models or functions relating x_n to y_n!

Gradient Boosting as Gradient Descent (cont.)

But why do we care that gradient boosting is gradient descent?

By making this connection, we can import the massive amount of techniques for studying gradient descent to analyze gradient boosting.

For example, we can easily reason about how to choose the learning rate λ in gradient boosting.

Choosing a Learning Rate

Under ideal conditions, gradient descent iteratively approximates and converges to the optimum.

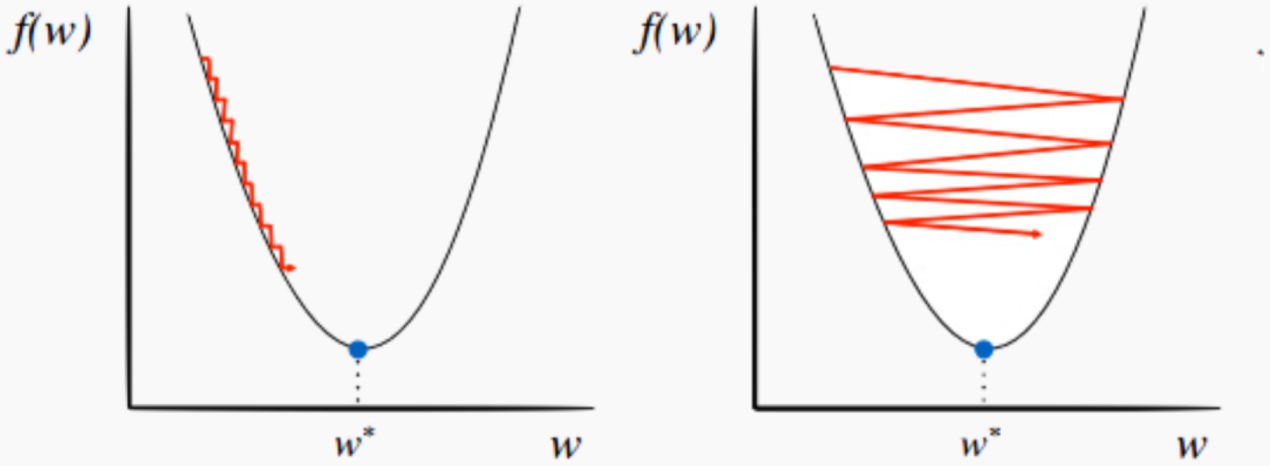
When do we terminate gradient descent?

- We can limit the number of iterations in the descent. But for an arbitrary choice of maximum iterations, we cannot guarantee that we are sufficiently close to the optimum in the end.
- If the descent is stopped when the updates are sufficiently small (e.g. the residuals of T are small), we encounter a new problem: the algorithm may never terminate!

Both problems have to do with the magnitude of the learning rate, λ.

Choosing a Learning Rate

For a constant learning rate, λ, if λ is too small, it takes too many iterations to reach the optimum.



If λ is too large, the algorithm may 'bounce' around the optimum and never get sufficiently close.

Choosing a Learning Rate

Choosing λ:

- If λ is a constant, then it should be tuned through cross validation.
- For better results, use a variable λ. That is, let the value of λ depend on the gradient

$$\lambda = h(\|\nabla f(x)\|),$$

where $\|\nabla f(x)\|$ is the magnitude of the gradient, $\nabla f(x)$. So

  - around the optimum, when the gradient is small, λ should be small
  - far from the optimum, when the gradient is large, λ should be larger

Motivation for AdaBoost

Using the language of gradient descent also allow us to connect gradient boosting for regression to a boosting algorithm often used for classification, AdaBoost.

In classification, we typically want to minimize the classification error:

$$\text{Error} = \frac{1}{N}\sum_{n=1}^{N} \mathbb{1}(y_n \neq \hat{y}_n), \quad \mathbb{1}(y_n \neq \hat{y}_n) = \begin{cases} 0, & y_n = \hat{y}_n \\ 1, & y_n \neq \hat{y}_n \end{cases}$$

Naively, we can try to minimize Error via gradient descent, just like we did for MSE in gradient boosting.

Unfortunately, Error is not differentiable with respect to the predictions, $\hat{y}$ ☹

Motivation for AdaBoost (cont.)

Our solution: we replace the Error function with a differentiable function that is a good indicator of classification error.

The function we choose is called ***exponential loss***

$$\text{ExpLoss} = \frac{1}{N}\sum_{n=1}^{N}\exp\left(-y_n\hat{y}_n\right),\ \ y_n \in \{-1, 1\}$$

Exponential loss is differentiable with respect to $\hat{y}_n$ and it is an upper bound of Error.

Gradient Descent with Exponential Loss

We first compute the gradient for ExpLoss:

$$\nabla \text{Exp} = [-y_1 \exp(-y_1 \hat{y}_1), \ldots, -y_N \exp(-y_N \hat{y}_N)]$$

It's easier to decompose each $y_n \exp(-y_n \hat{y}_n)$ as $w_n y_n$, where $w_n = \exp(-y_n \hat{y}_n)$.

This way, we see that the gradient is just a re-weighting applied the target values

$$\nabla \text{Exp} = [-w_1 y_1, \ldots, -w_N y_N]$$

Notice that when $y_n = \hat{y}_n$, the weight w_n is small; when $y_n \neq \hat{y}_n$, the weight is larger.

Gradient Descent with Exponential Loss

The update step in the gradient descent is

$$\hat{y}_n \leftarrow \hat{y}_n - \lambda w_n y_n, \quad n = 1, \ldots, N$$

Just like in gradient boosting, we approximate the gradient, $\lambda w_n y_n$ with a simple model, $T^{(i)}$, that depends on x_n.

This means training $T^{(i)}$ on a re-weighted set of target values,

$$\{(x_1, w_1 y_1), \ldots, (x_N, w_N, y_N)\}$$

That is, gradient descent with exponential loss means iteratively training simple models that ***focuses on the points misclassified by the previous model.***

AdaBoost

With a minor adjustment to the exponential loss function, we have the algorithm for gradient descent:

1. Choose an initial distribution over the training data, $w_n = 1/N$.
2. At the i^{th} step, fit a simple classifier $T^{(i)}$ on weighted training data

$$\{(x_1, w_1 y_1), ..., (x_N, w_N, y_N)\}$$

3. Update the weights:

$$w_n \leftarrow \frac{w_n \exp(-\lambda^{(i)} y_n T^{(i)}(x_n))}{Z}$$

where Z is the normalizing constant for the collection of updated weights

4. Update T: $T \leftarrow T + \lambda^{(i)} T^{(i)}$

where λ is the learning rate.

Choosing the Learning Rage

Unlike in the case of gradient boosting for regression, we can analytically solve for the optimal learning rate for AdaBoost, by optimizing:

$$\underset{\lambda}{\text{argmin}} \frac{1}{N} \sum_{n=1}^{N} \exp\left[-y_n(T + \lambda^{(i)} T^{(i)}(x_n))\right]$$

Doing so, we get that

$$\lambda^{(i)} = \frac{1}{2} \ln \frac{1-\epsilon}{\epsilon}, \quad \epsilon = \sum_{n=1}^{N} w_n \mathbb{1}(y_n \neq T^{(i)}(x_n))$$

Boosting in sklearn

Python has boosting algorithms implemented for you:

- **`sklearn.ensemble.AdaBoostClassifier`**
- **`sklearn.ensemble.AdaBoostRegressor`**
- With arguments of **base_estimator** (what models to use), **n_estimators** (max number of models to use), **learning_rate** (λ), etc…

Stacking

Motivation for Stacking

Recall that in boosting, the final model T, we learn is a weighted sum of simple models, T_h,

$$T = \sum_h \lambda_h T_H$$

where λ_h is the learning rate. In AdaBoost for example, we can analytically determine the optimal values of λ_h for each simple model T_h.

On the other hand, we can also determine the final model T implicitly by ***learning any model, called meta-learner, that transforms the outputs of T_h into a prediction***.

Stacked Generalization

The framework for ***stacked generalization*** or ***stacking*** (Wolpert 1992) is:

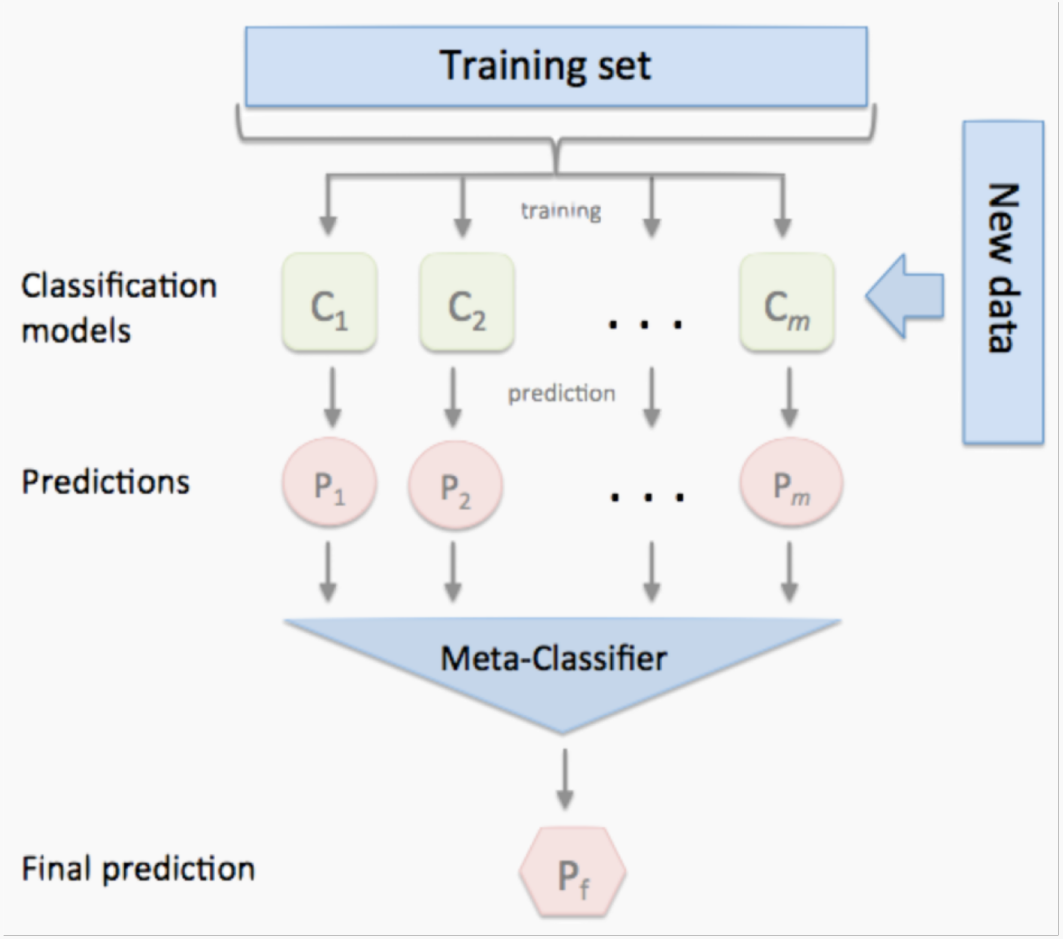
$$\{(x_1, y_1), ..., (x_N, y_N)\}$$

- train L number of models, T_l on the training data

$$\{(T_1(x_1), ..., T_L(x_1), y_1), ..., (T_1(x_N), ..., T_L(x_N), y_N)\}$$

- train a meta-learner $\tilde{T}$ on the predictions of the ensemble of models, i.e. train using the data

Stacking: an Illustration



Stacking: General Guidelines

The flexibility of stacking makes it widely applicable but difficult to analyze theoretically. Some general rules have been found through empirical studies:

- models in the ensemble should be diverse, i.e. their errors should not be correlated
- for binary classification, each model in the ensemble should have error rate < 1/2
- if models in the ensemble outputs probabilities, it's better to train the meta-learner on probabilities rather than predictions
- apply regularization to the meta-learner to avoid overfitting

Stacking: Subsemble Approach

We can extend the stacking framework to include ensembles of models that specialize on small subsets of data (Sapp et. al. 2014), for de-correlation or improved computational efficiency:

- divide the data in to J subsets
- train models, T_j , on each subset
- train a meta-learner $\tilde{T}$ on the predictions of the ensemble of models, i.e. train using the data

$$\{(T_1(x_1), ..., T_J(x_1), y_1), ..., (T_1(x_N), ..., T_J(x_N), y_N)\}$$

Again, we want to make sure that each $T_j(x_i)$ is an out of sample prediction.

Stacking in sklearn

Unfortunately, Python does not have stacking algorithms implemented for you ☹

So how can we do it?

We can set it up by 'manually' fitting several base models, take the outputs of those models, and fitting the meta model on the outputs of those base models.

It's a model on models!