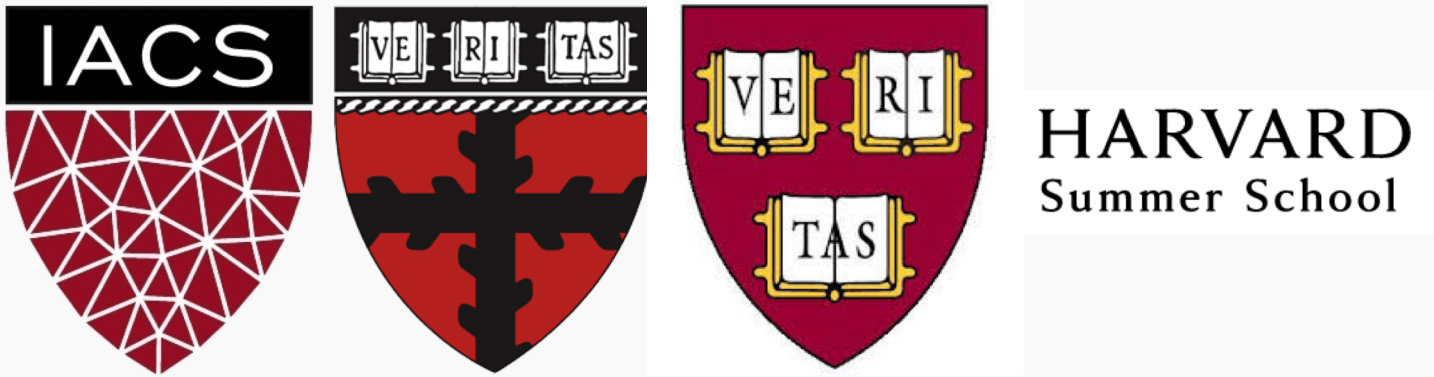
Lecture #7: *k*-NN Classification and PCA

CS-S109A: Introduction to Data Science

Kevin Rader



Lecture Outline

- Multiclass Classification
    - Multinomial and OvR Logistic Regression
- *k*-NN review
- *k*-NN for Classification
    - Multiclass *k*-NN
- Use of Interaction Terms and Unique Parameterizations
- Big Data and High Dimensionality
- Principal Components Analysis (PCA)

Logistic Regression for predicting 3+ Classes

There are several extensions to standard logistic regression when the response variable *Y* has more than 2 categories. The two most common are:

1. ordinal logistic regression
2. multinomial logistic regression.

Ordinal logistic regression is used when the categories have a specific hierarchy (like class year: Freshman, Sophomore, Junior, Senior; or a 7-point rating scale from strongly disagree to strongly agree).

Multinomial logistic regression is used when the categories have no inherent order (like eye color: blue, green, brown, hazel, et...).

Multinomial Logistic Regression

Multinomial Logistic Regression

There are two common approaches to estimating a nominal (not-ordinal) categorical variable that has more than 2 classes. The first approach sets one of the categories in the response variable as the *reference* group, and then fits separate logistic regression models to predict the other cases based off of the reference group. For example we could attempt to predict a student's concentration:

$$y = \begin{cases} 1 & if \text{ Computer Science (CS)} \\ 2 & if \text{ Statistics} \\ 3 & \text{otherwise} \end{cases}$$

from predictors X_1 number of psets per week, X_2 how much time playing video games per week, etc.

Multinomial Logistic Regression (cont.)

We could select the $y = 3$ case as the reference group (other concentration), and then fit two separate models: a model to predict $y = 1$ (CS) from $y = 3$ (others) and a separate model to predict $y = 2$ (Stat) from $y = 3$ (others).

Ignoring interactions, how many parameters would need to be estimated?

How could these models be used to estimate the probability of an individual falling in each concentration?

Multinomial Logistic Regression: the model

To predict K classes ($K > 2$) from a set of predictors X, a multinomial logistic regression can be fit:

$$\ln\left(\frac{P(Y=1)}{P(Y=K)}\right) = \beta_{0,1} + \beta_{1,1}X_1 + \beta_{2,1}X_2 + \cdots + \beta_{p,1}X_p$$

$$\ln\left(\frac{P(Y=2)}{P(Y=K)}\right) = \beta_{0,2} + \beta_{1,2}X_1 + \beta_{2,2}X_2 + \cdots + \beta_{p,2}X_p$$

$$\vdots$$

$$\ln\left(\frac{P(Y=K-1)}{P(Y=K)}\right) = \beta_{0,K-1} + \beta_{1,K-1}X_1 + \beta_{2,K-1}X_2 + \cdots + \beta_{p,K-1}X_p$$

Each separate model can be fit as independent standard logistic regression models!

Multinomial Logistic Regression in sklearn

```
mlogit = LogisticRegression(penalty='none',
                            multi_class='multinomial')
mlogit.fit(nfl_19[['ydstogo','down']],nfl_19['playtype'])
print(mlogit.intercept_)
print(mlogit.coef_)
```

```
[-6.78443446  4.46327069  2.32116377]
[[ 0.08130727  1.86026457]
 [-0.10687815 -1.33128781]
 [ 0.02557089 -0.52897676]]
```

```
pd.crosstab(nfl_19["play_type"],
            nfl_19["playtype"])
```

playtype	0	1	2
play_type			
field_goal	978	0	0
pass	0	0	18981
punt	2150	0	0
qb_kneel	393	0	0
qb_spike	72	0	0
run	0	12994	0

But wait Kevin, I thought you said we only fit $K - 1$ logistic regression models!?!? Why are there K intercepts and K sets of coefficients????

What is sklearn doing?

The *K* - 1 models in multinomial regression lead to the following probability predictions:

$$\ln\left(\frac{P(Y=k)}{P(Y=K)}\right) = \beta_{0,k} + \beta_{1,k}X_1 + \beta_{2,k}X_k + \cdots + \beta_{p,k}X_p$$

$$\vdots$$

$$P(Y=k) = P(Y=K)e^{\beta_{0,k}+\beta_{1,k}X_1+\beta_{2,k}X_k+\cdots+\beta_{p,k}X_p}$$

This give us *K* – 1 equations to estimate *K* probabilities for everyone. But probabilities add up to 1 ☺, so we are all set.

Sklearn then converts the above probabilities back into new betas (just like logistic regression, but the betas won't match):

$$\ln\left(\frac{P(Y=k)}{P(Y\neq k)}\right) = \beta'_{0,k} + \beta'_{1,k}X_1 + \beta'_{2,k}X_k + \cdots + \beta'_{p,k}X_p$$

One vs. Rest (OvR) Logistic Regression

One vs. Rest (OvR) Logistic Regression

An alternative multiclass logistic regression model in sklearn is called the 'One vs. Rest' approach, which is our second method.

If there are 3 classes, then 3 separate logistic regressions are fit, where the probability of each category is predicted over the rest of the categories combined. So for the concentration example, 3 models would be fit:

- a first model would be fit to predict CS from (Stat and Others) combined.
- a second model would be fit to predict Stat from (CS and Others) combined.
- a third model would be fit to predict Others from (CS and Stat) combined.

An example to predict play call from the NFL data follows...

One vs. Rest (OvR) Logistic Regression: the model

To predict K classes ($K > 2$) from a set of predictors X, a multinomial logistic regression can be fit:

$$\ln\left(\frac{P(Y=1)}{P(Y\neq 1)}\right) = \beta_{0,1} + \beta_{1,1}X_1 + \beta_{2,1}X_2 + \cdots + \beta_{p,1}X_p$$

$$\ln\left(\frac{P(Y=2)}{P(Y\neq 2)}\right) = \beta_{0,2} + \beta_{1,2}X_1 + \beta_{2,2}X_2 + \cdots + \beta_{p,2}X_p$$

$$\vdots$$

$$\ln\left(\frac{P(Y=K)}{P(Y\neq K)}\right) = \beta_{0,K} + \beta_{1,K}X_1 + \beta_{2,K}X_2 + \cdots + \beta_{p,K}X_p$$

Again, each separate model can be fit as independent standard logistic regression models!

Softmax

So how do we convert a set of probability estimates from separate models to one set of probability estimates?

The ***softmax*** function is used. That is, the weights are just normalized for each predicted probability. AKA, predict the 3 class probabilities from each of the 3 models, and just rescale so they add up to 1.

Mathematically that is:

$$P(y = k|\vec{x}) = \frac{e^{\vec{x}^T \hat{\vec{\beta}}_k}}{\sum_{j=1}^{K} e^{\vec{x}^T \hat{\vec{\beta}}_j}}$$

where $\vec{x}$ is the vector of predictors for that observation and $\hat{\vec{\beta}}_k$ are the associated logistic regression coefficient estimates.

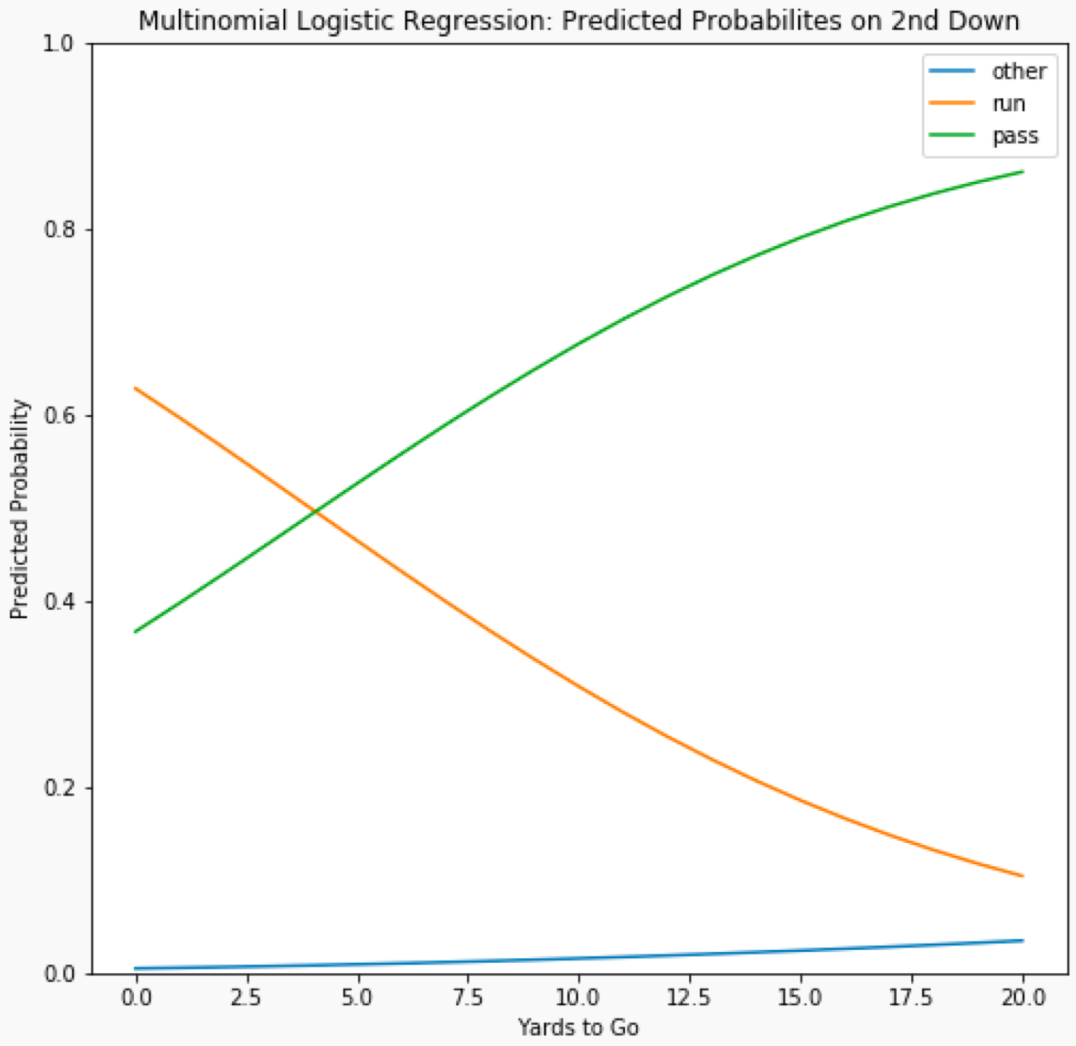
OVR Logistic Regression in Python

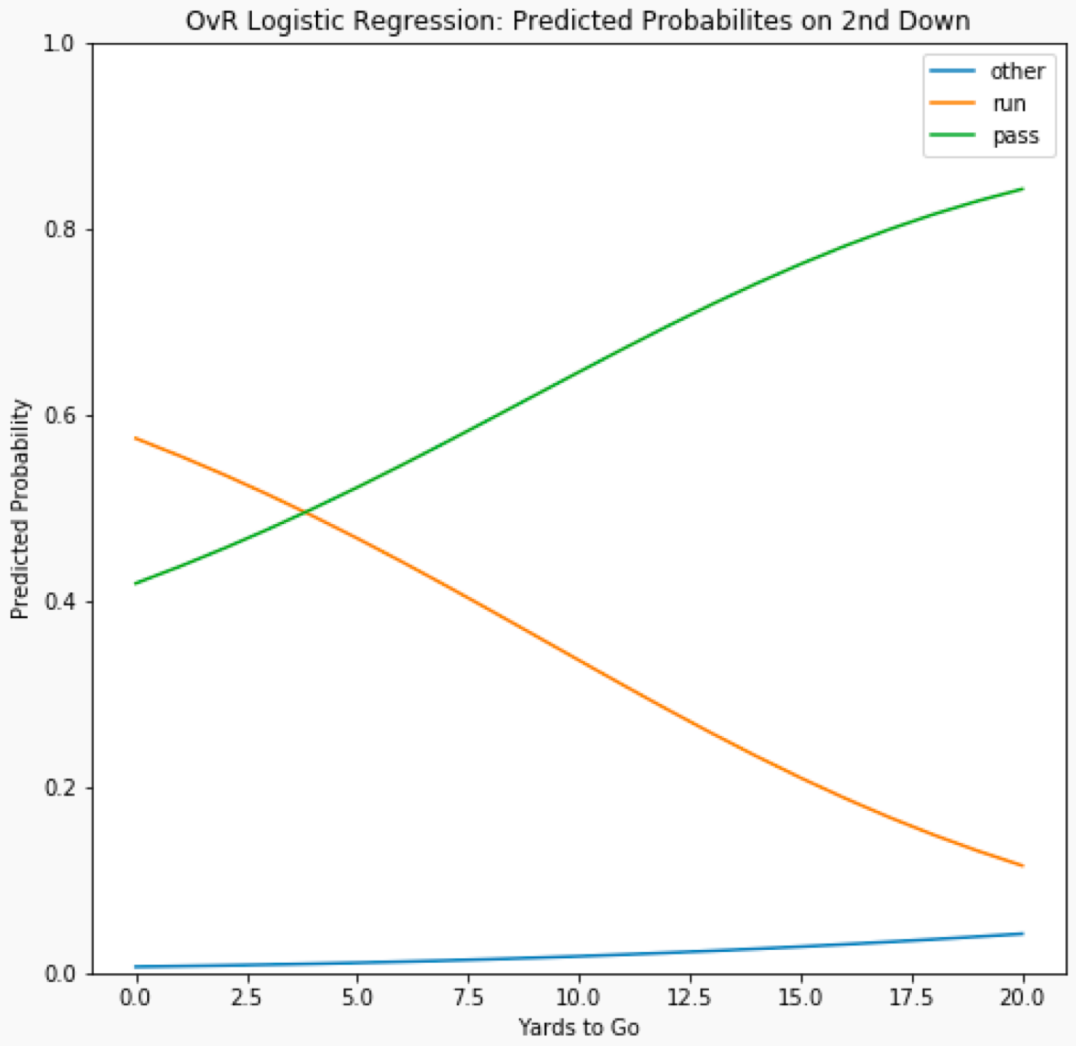
```
ovrlogit = LogisticRegression(penalty='none', multi_class='ovr')

ovrlogit.fit(nfl_19[['ydstogo','down']],nfl_19['playtype'])
print(ovrlogit.intercept_)
print(ovrlogit.coef_)
```

```
[-10.03308293   2.47802369  -0.10976034]
[[ 0.07547391  2.55126265]
 [-0.14272983 -0.98841346]
 [ 0.03672441 -0.03755844]]
```

Phew! This one is as expected ☺

Predicting Type of Play in the NFL





Classification for more than 2 Categories

When there are more than 2 categories in the response variable, then there is no guarantee that $P(Y = k) \geq 0.5$ for any one category. So any classifier based on logistic regression (or other classification model) will instead have to select the group with **the largest estimated probability**.

The classification boundaries are then much more difficult to determine mathematically. We will not get into the algorithm for determining these in this class, but we can rely on `predict` and `predict_proba`!

Prediction using Multiclass Logistic Regression

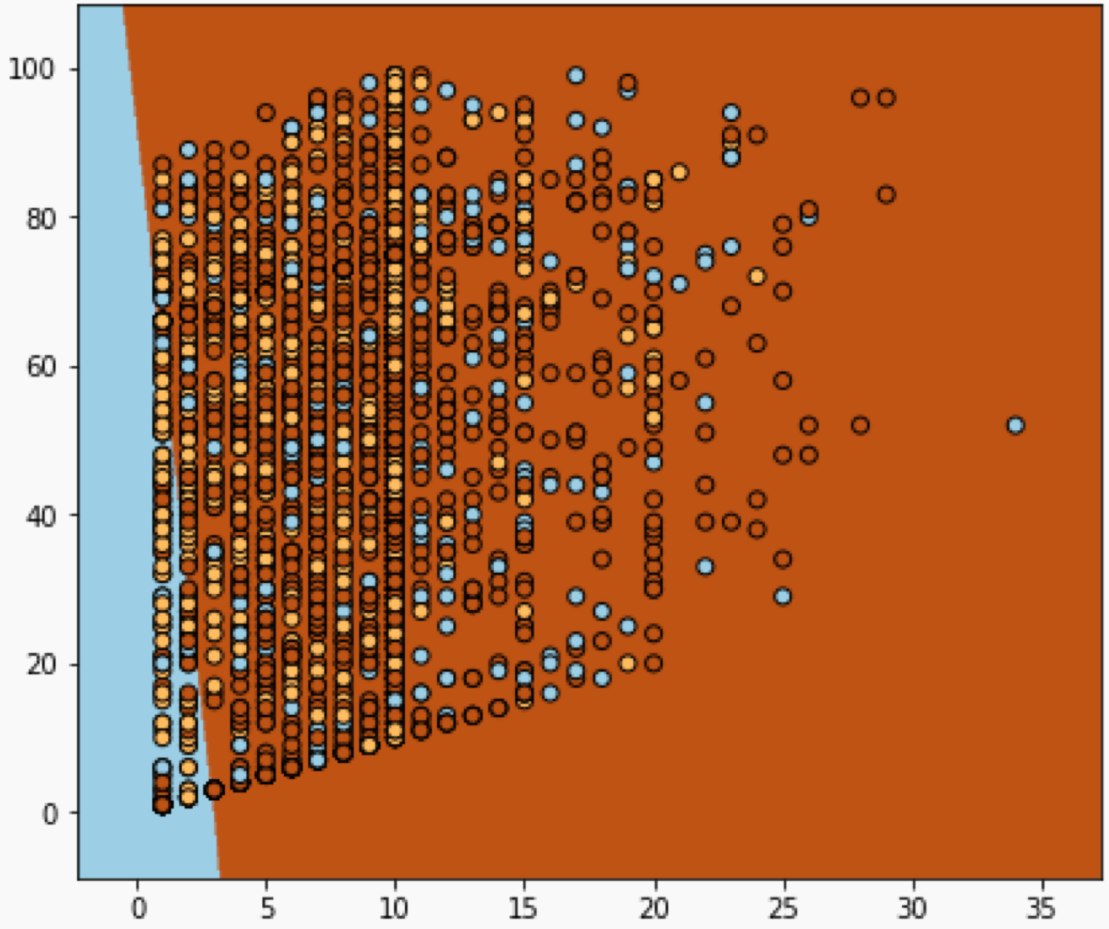
```
print(mlogit.predict_proba(nfl_19[['ydstogo','down']])[0:5,:])
print(mlogit.predict(nfl_19[['ydstogo','down']])[0:5])
```

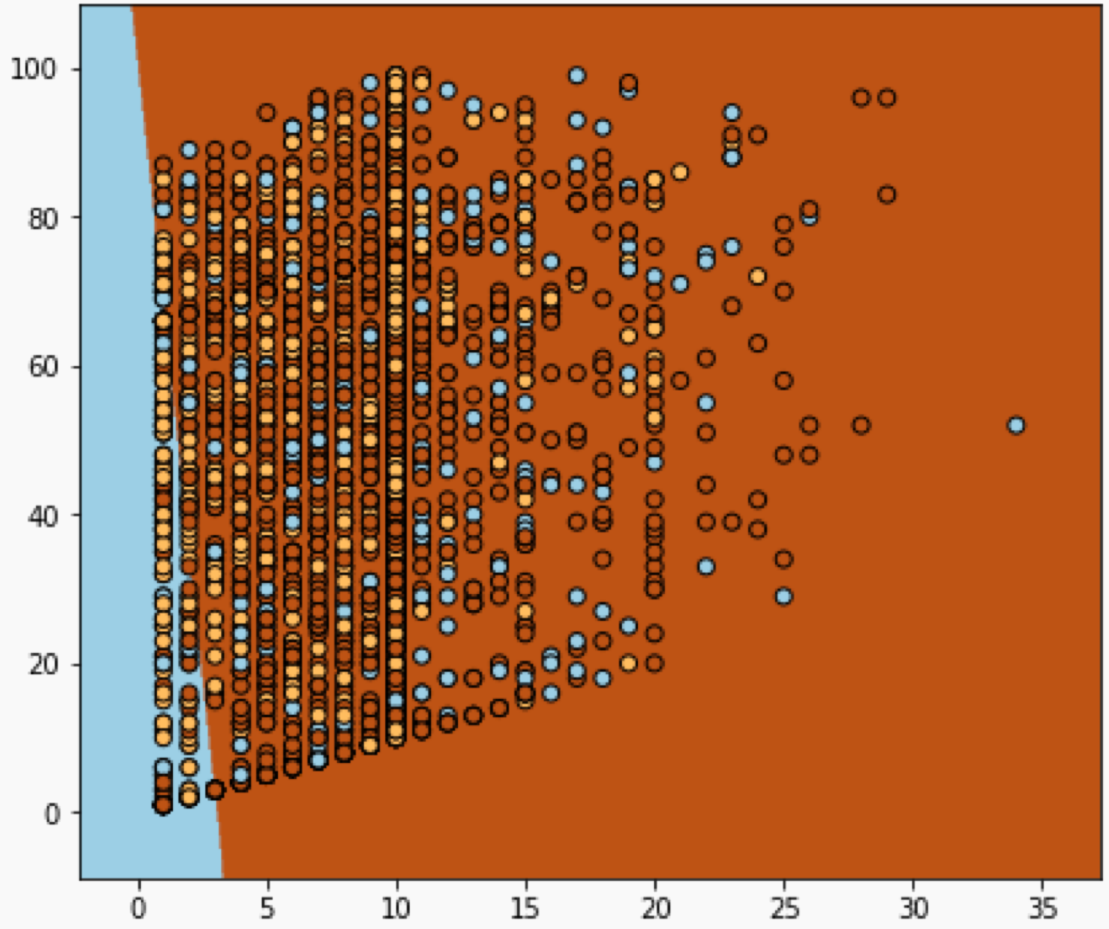
```
[[0.001048   0.50329827 0.49565372]
 [0.01559789 0.30793278 0.67646934]
 [0.17275682 0.14020122 0.68704196]
 [0.82376365 0.00418569 0.17205066]
 [0.001048   0.50329827 0.49565372]]
[1 2 2 0 1]
```

```
print(ovrlogit.predict_proba(nfl_19[['ydstogo','down']])[0:5,:])
print(ovrlogit.predict(nfl_19[['ydstogo','down']])[0:5])
```

```
[[0.00111671 0.48115571 0.51772757]
 [0.01792012 0.33603242 0.64604746]
 [0.19847963 0.15493954 0.64658083]
 [0.57254676 0.0088239  0.41862935]
 [0.00111671 0.48115571 0.51772757]]
[2 2 2 0 2]
```

Classification Boundary for 3+ Classes in sklearn





Estimation and Regularization in multiclass settings

There is no difference in the approach to estimating the coefficients in the multiclass setting: we maximize the log-likelihood (or minimize negative log-likelihood).

This combined negative log-likelihood of all *K* classes is sometimes called the **binary cross-entropy**:

$$\ell = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} \mathbb{1}(y_i = k)\ln(\hat{P}\,(y_i = k)) + \mathbb{1}(y_i \neq k)\ln(1 - \hat{P}\,(y_i = k))$$

And regularization can be done like always: add on a penalty term to this loss function based on L1 (sum of the absolute values) or L2 (sum of squares) norms.

k-NN for Classification

k-NN for Classification

How can we modify the *k*-NN approach for classification?

The approach here is the same as for *k*-NN regression: use the other available observations that are most similar to the observation we are trying to predict (classify into a group) based on the predictors at hand.

How do we classify which category a specific observation should be in based on its nearest neighbors?

The category that shows up the most among the nearest neighbors.

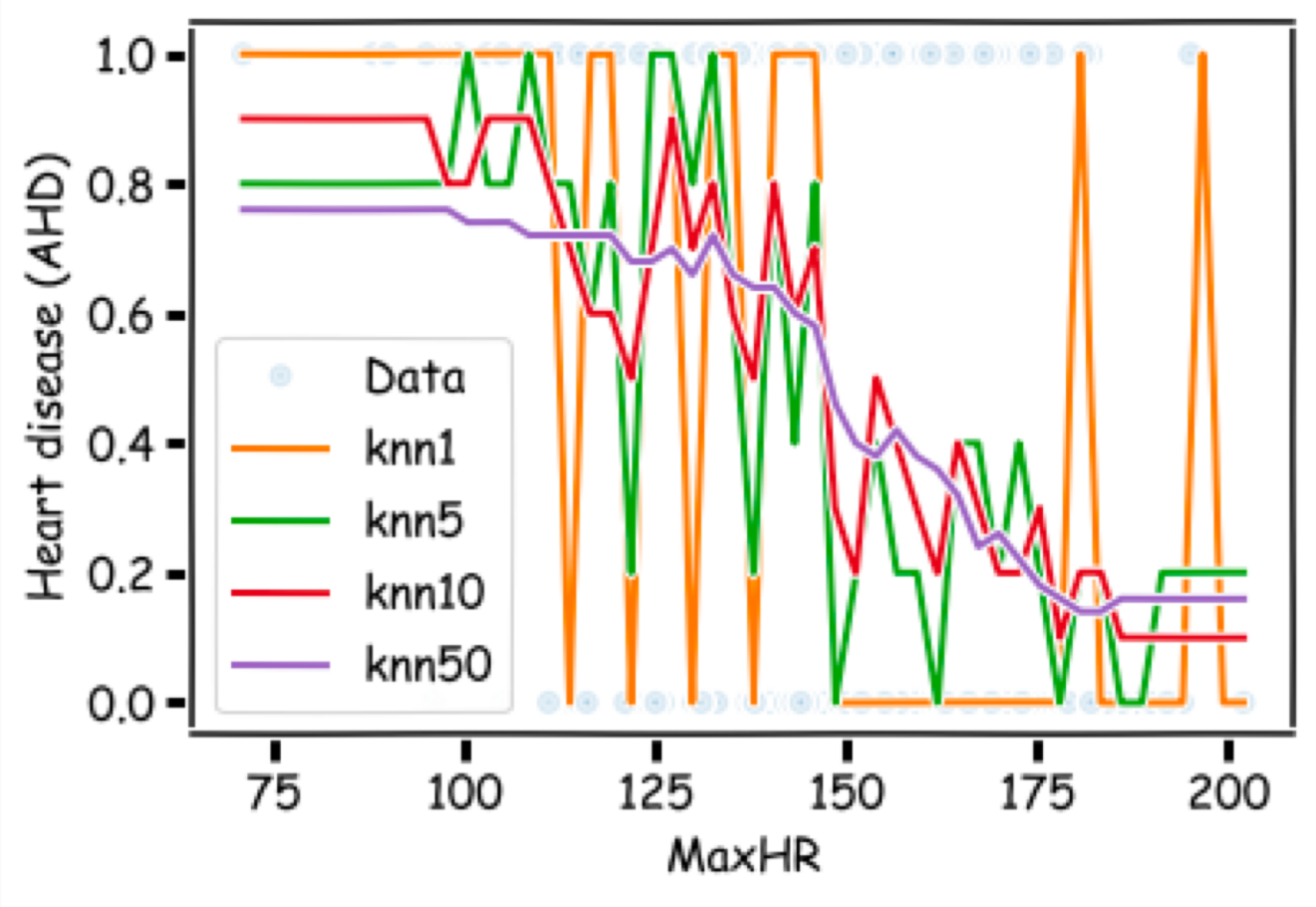
k-NN for Classification: formal definition

The *k*-NN classifier first identifies the *k* points in the training data that are closest to x_0, represented by $\mathcal{N}_0$. It then estimates the conditional probability for class *j* as the fraction of points in $\mathcal{N}_0$ whose response values equal *j*:

$$P(Y = j | X = x_0) = \frac{1}{k} \sum_{i \in \mathcal{N}_0} I(y_i = j)$$

Then, the *k*-NN classifier applies Bayes rule and classifies the test observation, x_0, to the class with largest estimated probability.

Estimated Probabilities in *k*-NN Classification



k-NN for Classification (cont.)

There are some issues that may arise:

- How can we handle a tie?

- What could be a major problem with always classifying to the most common group amongst the neighbors?

- How can we handle this?

k-NN with Multiple Predictors

How could we extend *k*-NN (both regression and classification) when there are multiple predictors?

We would need to define a measure of distance for observations in order to which are the most similar to the observation we are trying to predict.

Euclidean distance is a good option. To measure the distance of a new observation, $\boldsymbol{x}_0$ from each observation in the data set, $\boldsymbol{x}_i$:

$$D^2(\boldsymbol{x}_i, \boldsymbol{x}_0) = \sum_{j=1}^{P} \left(x_{i,j} - x_{0,j}\right)^2$$

k-NN with Multiple Predictors (cont.)

But what must we be careful about when measuring distance?

1. Differences in variability in our predictors!
2. Having a mixture of quantitative and categorical predictors.

So what should be good practice? To determine closest neighbors when $p > 1$, you should first standardize the predictors! And you can even standardize the binaries if you want to include them.

How else could we determine closeness in this multi-dimensional setting?

k-NN Classification in Python

Performing kNN classification in python is done via **`KNeighborsClassifier`** in **`sklearn.neighbors`**.

An example:

```
#two predictors
from sklearn import neighbors

knn1 = neighbors.KNeighborsClassifier(n_neighbors=1)
knn5 = neighbors.KNeighborsClassifier(n_neighbors=5)
knn10 = neighbors.KNeighborsClassifier(n_neighbors=10)
knn50 = neighbors.KNeighborsClassifier(n_neighbors=50)

data_x = df_heart[['MaxHR','RestBP']]
data_y = df_heart.AHD.map(lambda x: 0 if x=='No' else 1)

knn1.fit(data_x, data_y);
knn5.fit(data_x, data_y);
knn10.fit(data_x, data_y);
knn50.fit(data_x, data_y);

print(knn1.score(data_x, data_y))
print(knn5.score(data_x, data_y))
print(knn10.score(data_x, data_y))
print(knn50.score(data_x, data_y))
```

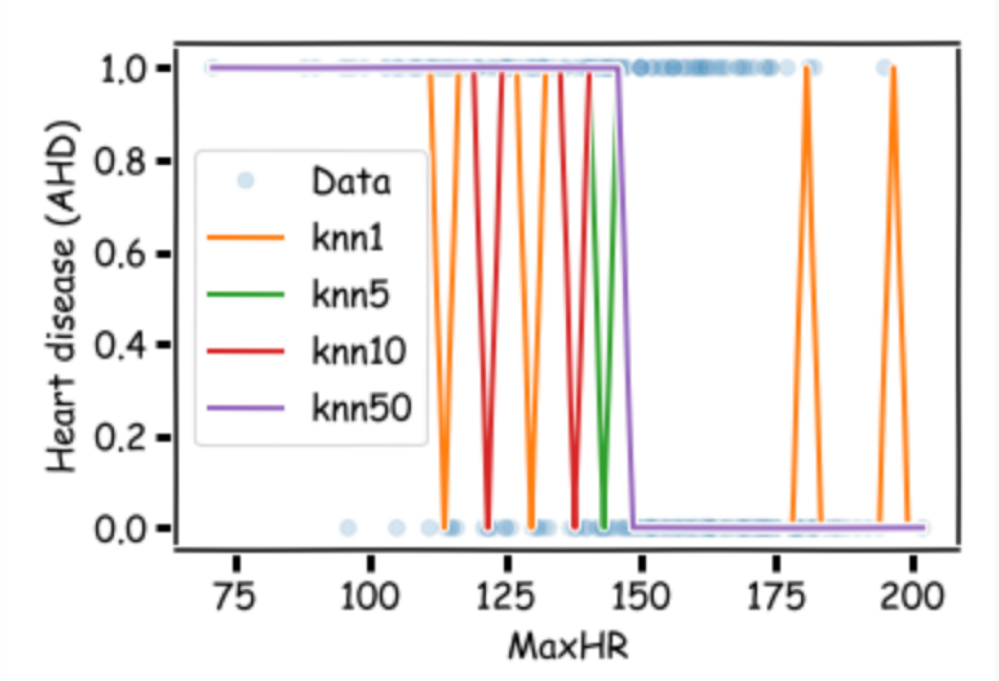
```
0.960396039604
0.712871287129
0.716171617162
0.706270627063
```

Classification Boundaries in *k*-NN Classification

What will the classification boundaries look line in *k*-NN classification? With one predictors? With 2 predictors? With 3+ predictors?

How can we visualize these? In 1D? In 2D? In 3D+?

How do they compare to Logistic Regression classification boundaries?



Multiclass *k*-NN

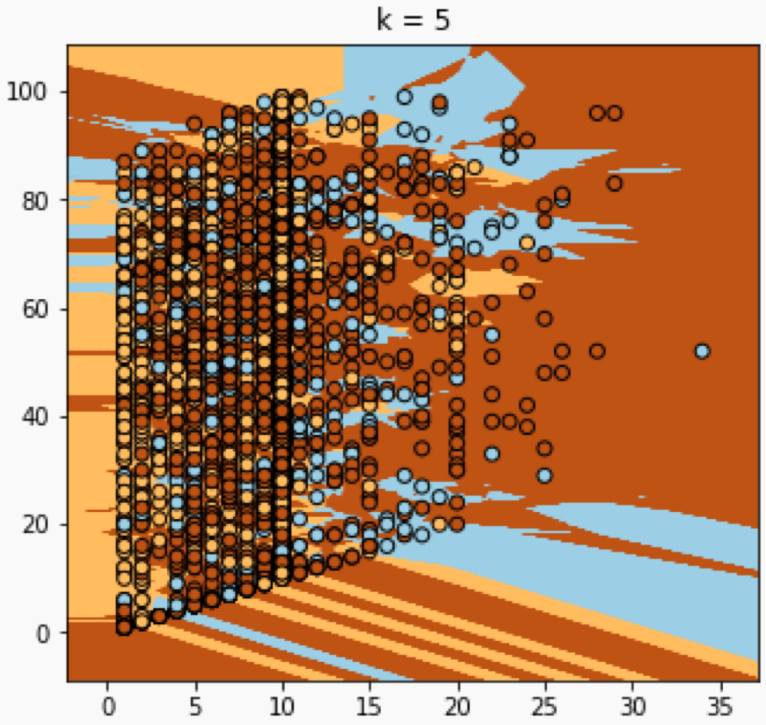
k-NN for 3+ Classes

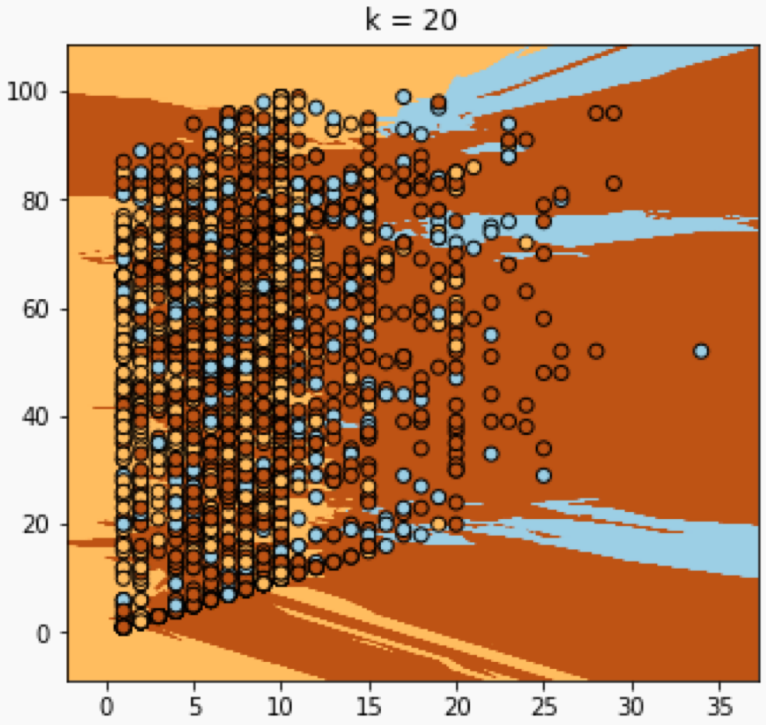
Extending the *k*-NN classification model to 3+ classes is much simpler!

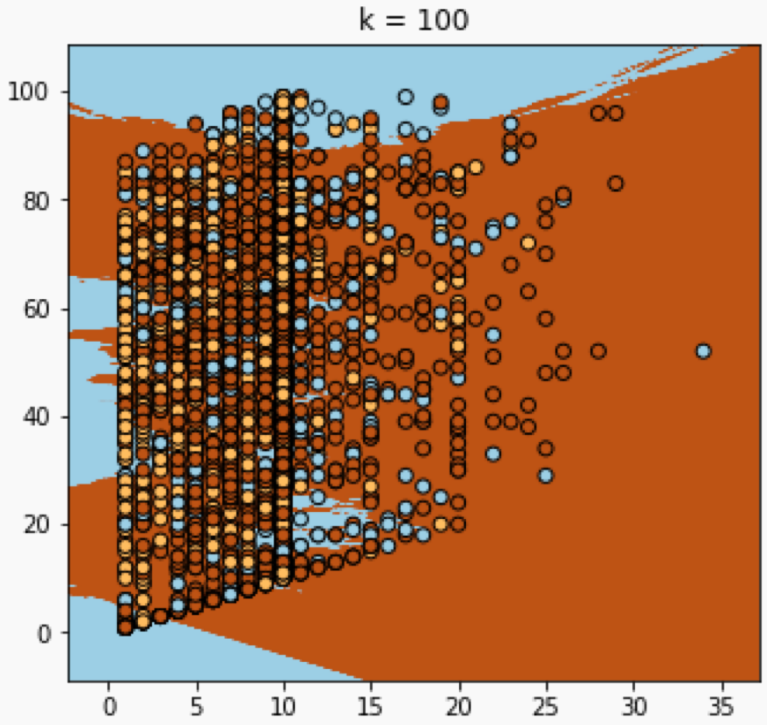
Remember, *k*-NN is done in two steps:

1. **Find you *k* neighbors**: this is still done in the exact same way
   - be careful of the scaling of your predictors!
2. **Predict based on your neighborhood:**
   - Predicting probabilities: just use the observed proportions
   - Predicting classes: plurality wins!

k-NN for 3+ Classes: NFL Data







k-Nearest Neighbors

We've already seen the *k*-NN method for predicting a quantitative response (it was the very first method we introduced). How was *k*-NN implemented in the Regression setting (quantitative response)?

The approach was simple: to predict an observation's response, use the **other** available observations that are most similar to it.

For a specified value of *k*, each observation's outcome is predicted to be the **average** of the *k*-closest observations as measured by some distance of the predictor(s).

With one predictor, the method was easily implemented.

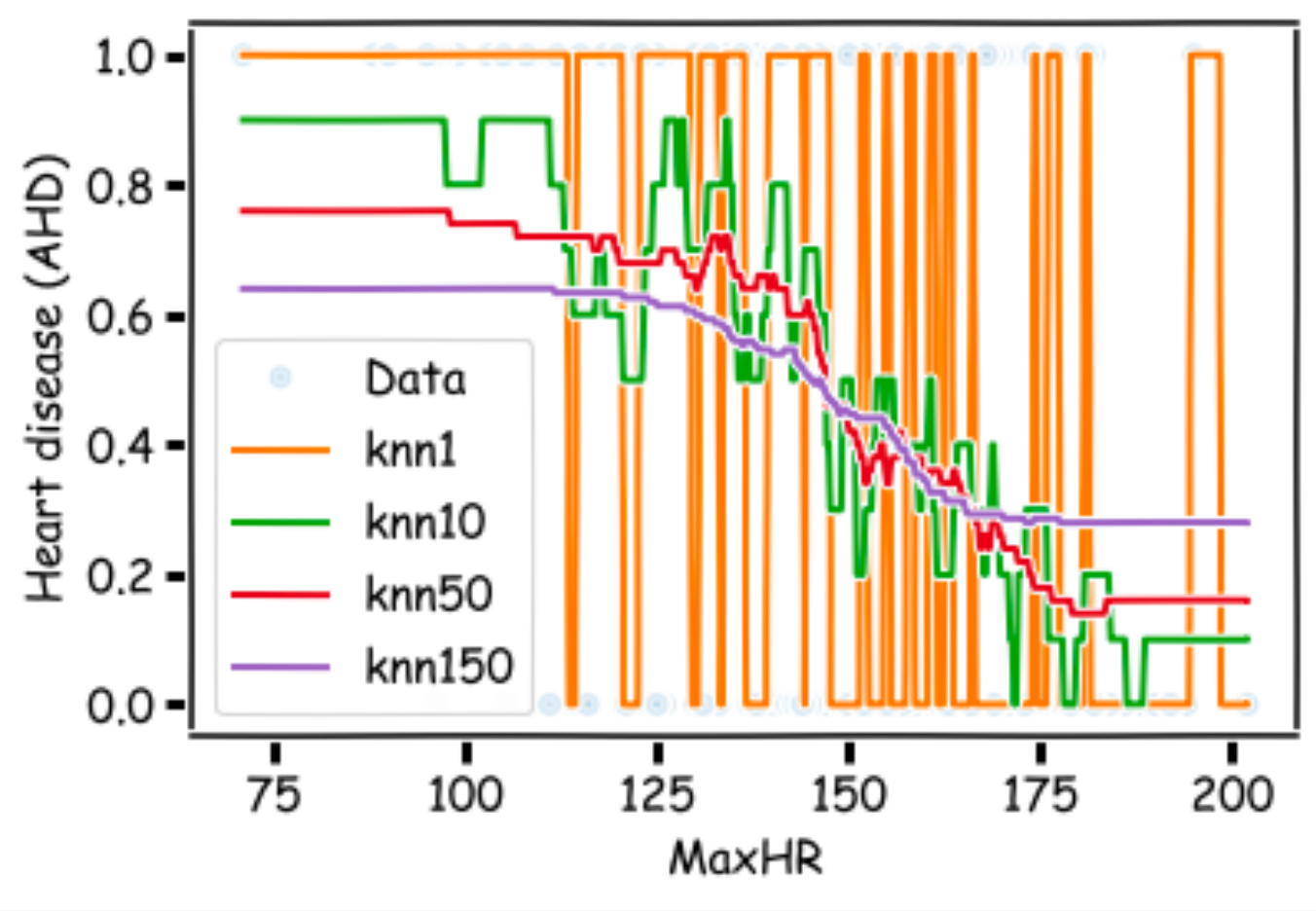
Review: Choice of k (now for classification)

How well the predictions perform is related to the choice of k.

What will the predictions look like if k is very small? What if it is very large?

More specifically, what will the predictions be for new observations if $k = n$?

Choice of *k* matters



Interaction Terms and Unique Parameterizations

NYC Taxi vs. Uber

We'd like to compare Taxi and Uber rides in NYC (for example, how much the fare costs based on length of trip, time of day, location, etc.).

A public dataset has 1.9 million Taxi and Uber trips. Each trip is described by p = 23 useable predictors (and 1 response variable).

```
In [11]: print(nyc_cab_df.shape)
         nyc_cab_df.head()

         (1873671, 30)
```

Out[11]:

	AWND	Base	Day	Dropoff_latitude	Dropoff_longitude	Ehail_fee	Extra	Fare_amount	Lpep_dropoff_datetime	MTA_tax	...	TMIN	Tip_amount	Tolls_amou
0	4.7	B02512	1	NaN	NaN	NaN	NaN	33.863498	2014-04-01 00:24:00	NaN	...	39	NaN	Na
1	4.7	B02512	1	NaN	NaN	NaN	NaN	19.022892	2014-04-01 00:29:00	NaN	...	39	NaN	Na
2	4.7	B02512	1	NaN	NaN	NaN	NaN	25.498981	2014-04-01 00:34:00	NaN	...	39	NaN	Na
3	4.7	B02512	1	NaN	NaN	NaN	NaN	28.024628	2014-04-01 00:39:00	NaN	...	39	NaN	Na
4	4.7	B02512	1	NaN	NaN	NaN	NaN	12.083589	2014-04-01 00:40:00	NaN	...	39	NaN	Na

5 rows × 30 columns

Interaction Terms: A Review

Recall that an interaction term between predictors X_1 and X_2 can be incorporated into a regression model by including the multiplicative (i.e. cross) term in the model, for example

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 (X_1 * X_2) + \varepsilon$$

Suppose X_1 is a binary predictor indicating whether a NYC ride pickup is a taxi or an Uber, X_2 is the length of the trip, and Y is the fare for the ride.

What is the interpretation of β_3?

Including Interaction Terms in Models

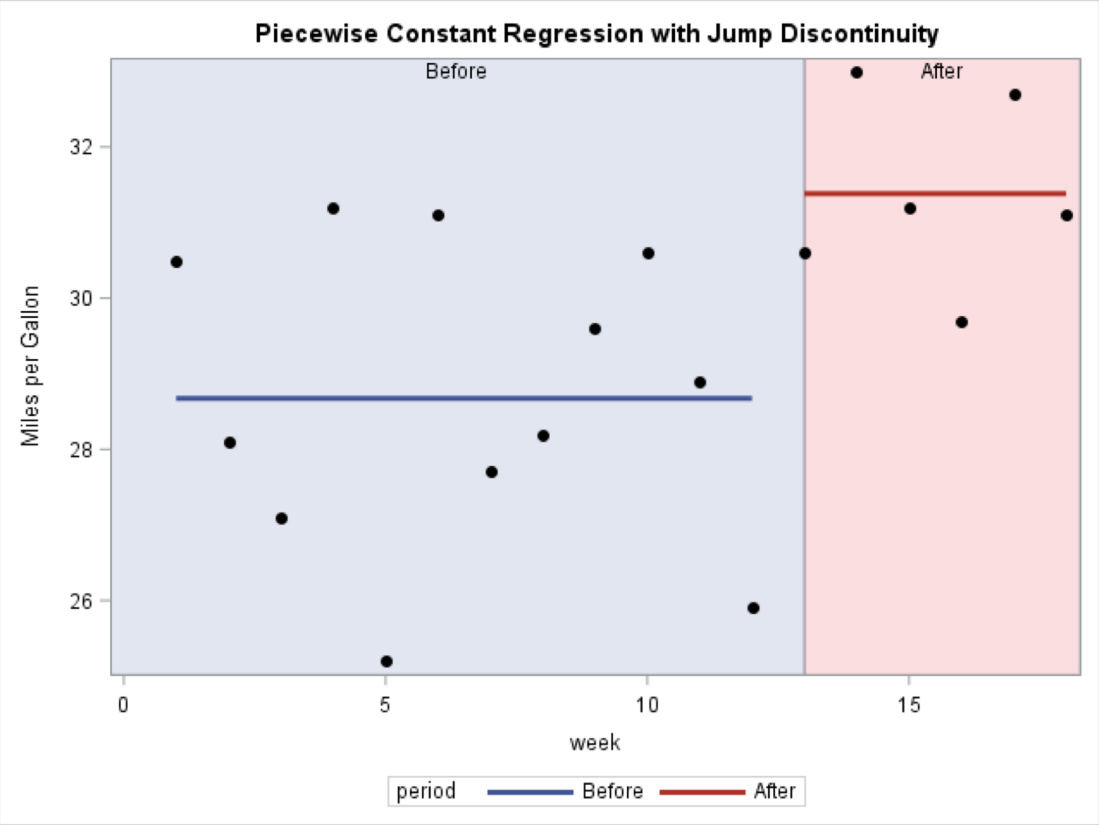
Recall that to avoid overfitting, we sometimes elect to exclude a number of terms in a linear model.

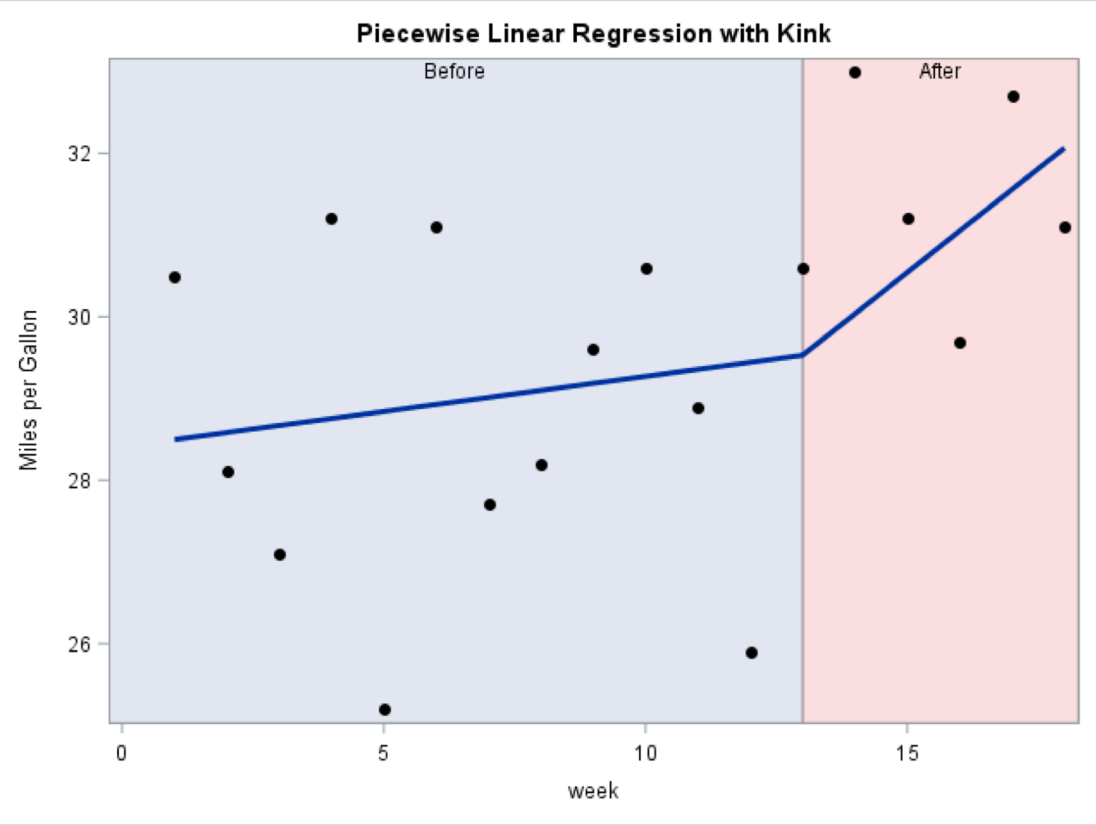
It is standard practice to always include the ***main effects*** in the model. That is, we always include the terms involving only one predictor, $\beta_1 X_1$, $\beta_2 X_2$ etc.

Question: Why are the ***main effects*** important?

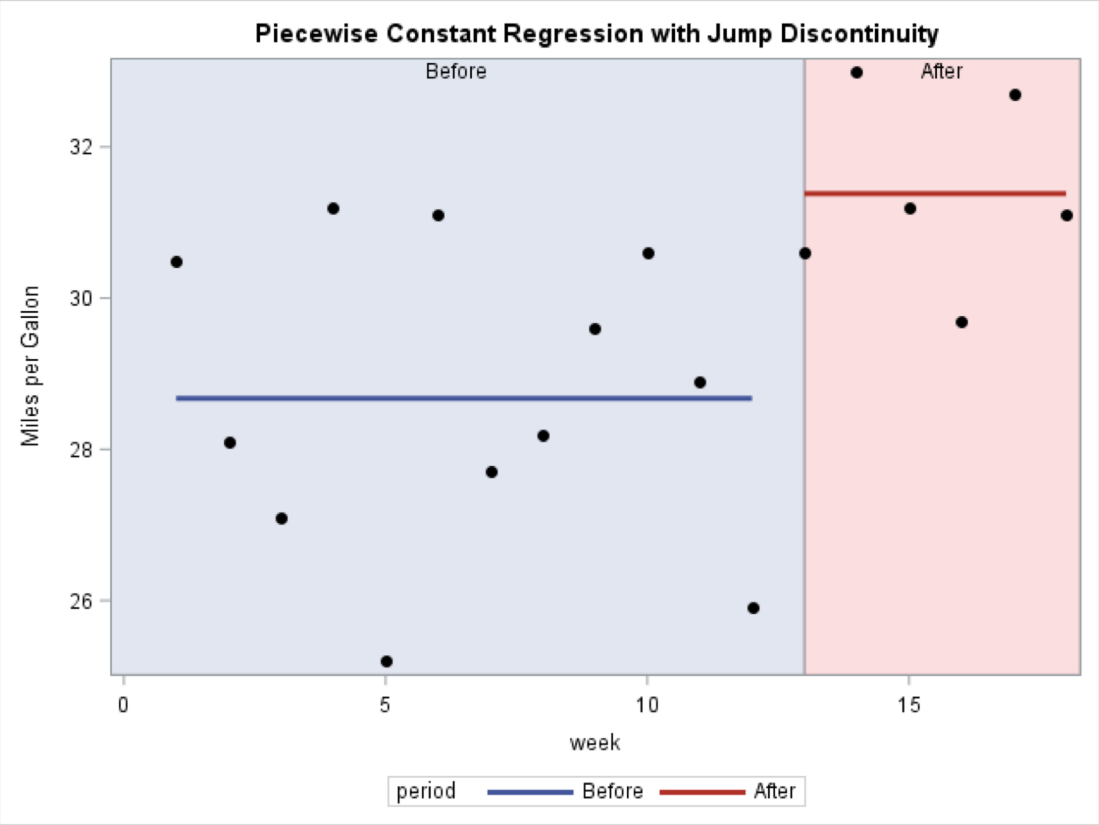
Question: In what type of model would it make sense to include the interaction term without one of the main effects?

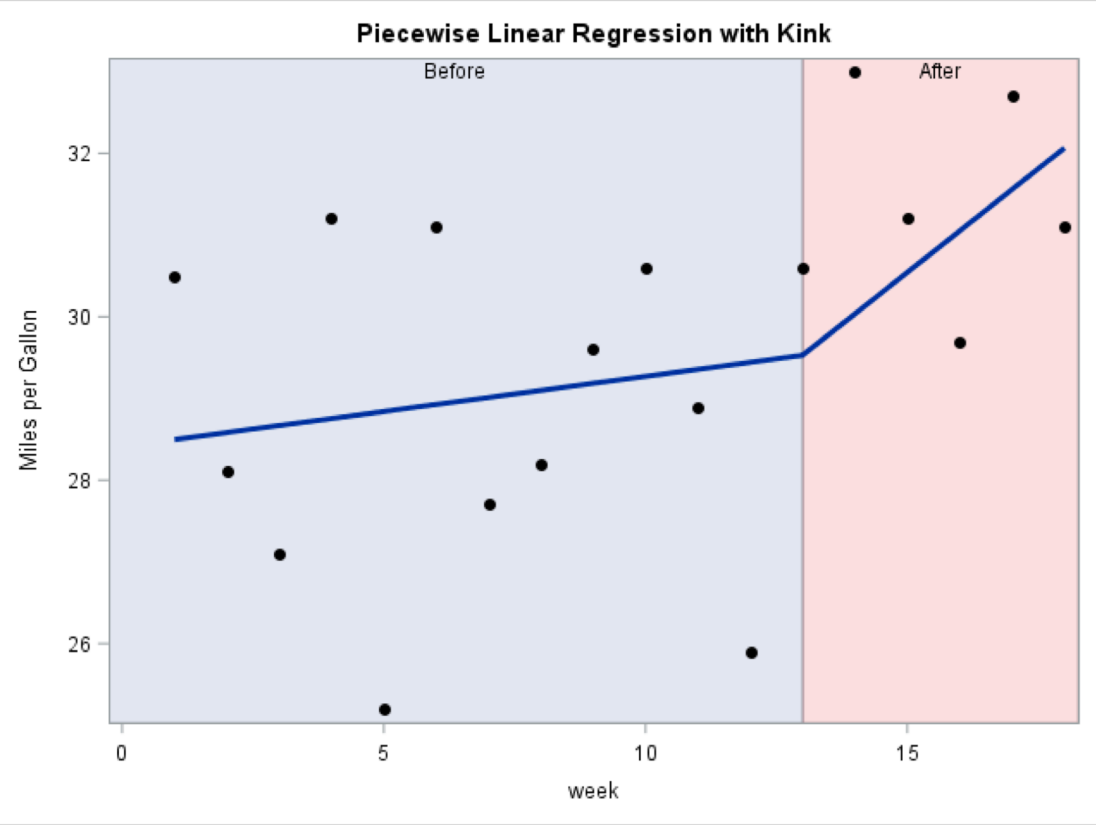
How would you *parameterize* these model?





How would you *parameterize* these model?





$$\hat{Y} = \beta_0 + \beta_1 \cdot I(X \geq 13)$$

$$\hat{Y} = \beta_0 + \beta_1 X + \beta_2 \cdot (X - 13) \cdot I(X \geq 13))$$

How Many Interaction Terms?

This NYC taxi and Uber dataset has 1.9 million Taxi and Uber trips. Each trip is described by p = 23 useable predictors (and 1 response variable). How many interaction terms are there?

- Two-way interactions: $\binom{p}{2} = \frac{p(p-1)}{2} = 253$
- Three-way interactions: $\binom{p}{3} = \frac{p(p-1)(p-2)}{6} = 1771$
- Etc.

The total number of all possible interaction terms (including main effects) is.

$$\sum_{k=0}^{p} \binom{p}{k} = 2^p \approx 8.3\text{million}$$

What are some problems with building a model that includes all possible interaction terms?

How Many Interaction Terms?

In order to wrangle a data set with roughly 2 million observations, we could use random samples of 100k observations from the dataset to build our models. If we include all possible interaction terms, our model will have 8.3 mil parameters. **We will not be able to uniquely determine 8.3 mil parameters with only 100k observations**. In this case, we call the model ***unidentifiable***.

To handle this in practice, we can:

- increase the number of observation
- consider only scientifically important interaction terms
- Use an appropriate method that account for this issue
- perform another ***dimensionality reduction*** technique like PCA

Big Data and High Dimensionality

What is ‘Big Data’?

In the world of Data Science, the term *Big Data* gets thrown around a lot. What does *Big Data* mean?

A rectangular data set has two dimensions: number of observations (*n*) and the number of predictors (*p*). Both can play a part in defining a problem as a *Big Data* problem.

What are some issues when:

- *n* is big (and *p* is small to moderate)?
- *p* is big (and *n* is small to moderate)?
- *n* and *p* are both big?

When *n* is big

When the sample size is large, this is typically not much of an issue from the statistical perspective, just one from the computational perspective.

- Algorithms can take forever to finish. Estimating the coefficients of a regression model, especially one that does not have closed form (like LASSO), can take a while. Wait until we get to Neural Nets!
- If you are tuning a parameter or choosing between models (using CV), this exacerbates the problem.

What can we do to fix this computational issue?

- Perform 'preliminary' steps (model selection, tuning, etc.) on a subset of the training data set. 10% or less can be justified

Keep in mind, big *n* doesn't solve everything

The era of Big Data (aka, large *n*) can help us answer lots of interesting scientific and application-based questions, but it does not fix everything.

Remember the old adage: "**crap in = crap out**". That is to say, if the data are not representative of the population, then modeling results can be terrible. Random sampling ensures representative data.

Xiao-Li Meng does a wonderful job describing the subtleties involved (WARNING: it's a little technical, but digestible):
https://www.youtube.com/watch?v=8YLdIDOMEZs

When *p* is big

When the number of predictors is large (in any form: interactions, polynomial terms, etc.), then lots of issues can occur.

- Matrices may not be invertible (issue in OLS).
- Multicollinearity is likely to be present
- Models are susceptible to overfitting

This situation is called *High Dimensionality*, and needs to be accounted for when performing data analysis and modeling.

What techniques have we learned to deal with this?

When Does High Dimensionality Occur?

The problem of high dimensionality can occur when the number of parameters exceeds or is close to the number of observations. This can occur when we consider lots of interaction terms, like in our previous example. But this can also happen when the number of main effects is high.

For example:

- When we are performing polynomial regression with a high degree and a large number of predictors.
- When the predictors are genomic markers (and possible interactions) in a computational biology problem.
- When the predictors are the counts of all English words appearing in a text.

How Does sklearn handle unidentifiability?

In a parametric approach: if we have an over-specified model ($p > n$), the parameters are unidentifiable: we only need $n - 1$ predictors to perfectly predict every observation ($n - \mathbf{1}$ because of the intercept).

So what happens to the 'extra' parameter estimates (the extra β's)?

- the remaining $p - (n - 1)$ predictors' coefficients can be estimated to be anything. Thus there are an infinite number of sets of estimates that will give us identical predictions. There is not one unique set of $\hat{\beta}$'s!

What would be reasonable ways to handle this situation? How does sklearn handle this? When is another situation in which the parameter estimates are unidentifiable? What is the simplest case?

Perfect Multicollinearity

The $p > n$ situation leads to perfect collinearity of the predictor set. But this can also occur with a redundant predictors (ex: putting X_j twice into a model). Let's see what sklearn in this simplified situation:

```
In [9]: # investigating what happens when two identical predictors are used

        logit1 = LogisticRegression(C=1000000,solver="lbfgs").fit(heart_df[['Age']],y)
        logit2 = LogisticRegression(C=1000000,solver="lbfgs").fit(heart_df[['Age','Age']],y)

        print("The coef estimate for Age (when in the model once):",logit1.coef_)
        print("The coef estimates for Age (when in the model twice):",logit2.coef_)

        The coef estimate for Age (when in the model once): [[0.05198618]]
        The coef estimates for Age (when in the model twice): [[0.02599311 0.02599311]]
```

How does this generalize into the high-dimensional situation?

A Framework For Dimensionality Reduction

One way to reduce the dimensions of the feature space is to create a new, smaller set of predictors by taking linear combinations of the original predictors.

We choose $Z_1, Z_2, \ldots, Z_m$, where $m \leq p$ and where each Z_i is a linear combination of the original p predictors

$$Z_i = \sum_{j=1}^{p} \phi_{ji} X_j$$

for fixed constants ϕ_{ji}. Then we can build a linear regression regression model using the new predictors

$$Y = \beta_0 + \beta_1 Z_1 + \cdots + \beta_m Z_m + \varepsilon$$

Notice that this model has a smaller number (m+1 < p+1) of parameters.

A Framework For Dimensionality Reduction (cont.)

A method of dimensionality reduction includes 2 steps:

- Determine an optimal set of new predictors $Z_1,\ldots, Z_m$, for $m < p$.
- Express each observation in the data in terms of these new predictors. The transformed data will have m columns rather than p.

Thereafter, we can fit a model using the new predictors.

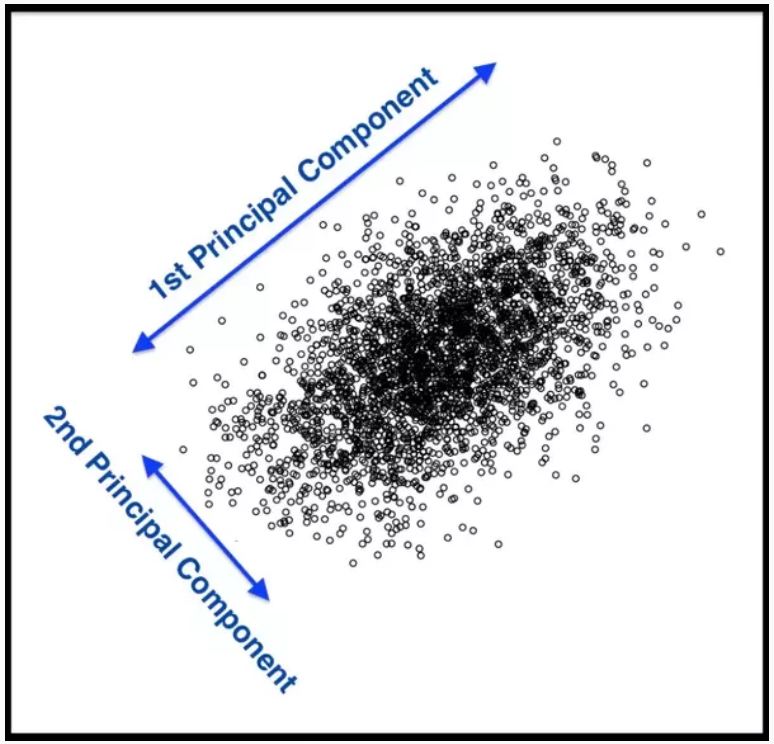
The method for determining the set of new predictors (what do we mean by an optimal predictors set?) can differ according to application. We will explore a way to create new predictors that captures the *essential* variations in the observed predictor set.

You’re Gonna Have a Bad Time…

Principal Components Analysis (PCA)

Principal Components Analysis (PCA)

Principal Components Analysis (PCA) is a method to identify a new set of predictors, as linear combinations of the original ones, that captures the `maximum amount' of variance in the observed data.



PCA (cont.)

Principal Components Analysis (PCA) produces a list of p **principal components** $Z_1,\ldots, Z_p$ such that

- Each Z_i is a linear combination of the original predictors, and it's vector norm is 1
- The Z_i's are pairwise orthogonal
- The Z_i's are ordered in decreasing order in the amount of captured observed variance.

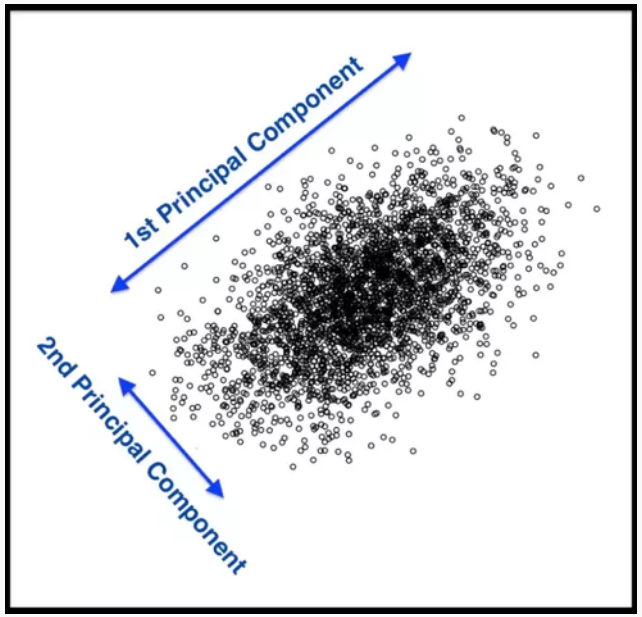
That is, the observed data shows more variance in the direction of Z_1 than in the direction of Z_2.

To perform dimensionality reduction we select the top m principle components of PCA as our new predictors and express our observed data in terms of these predictors.

The Intuition Behind PCA

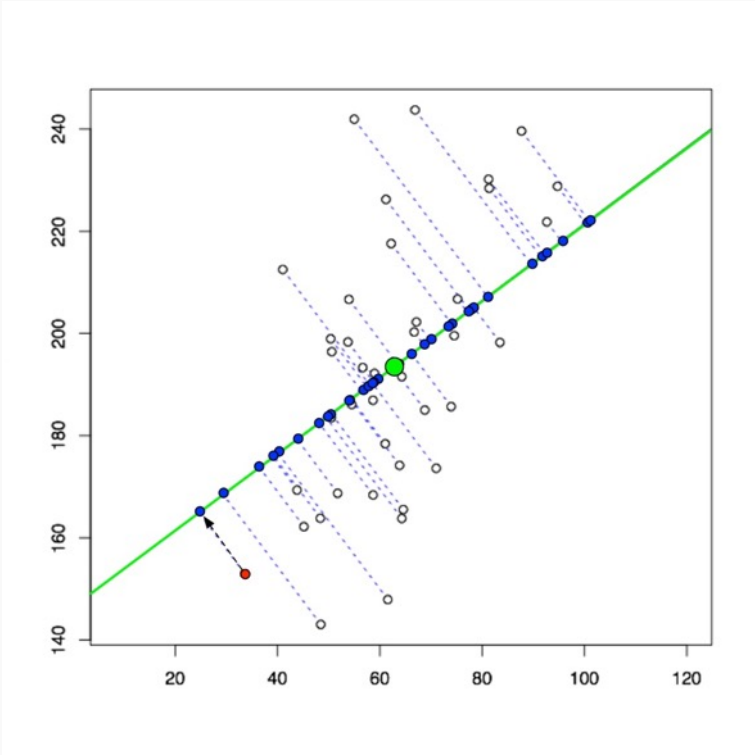
Top PCA components capture the most of amount of variation (interesting features) of the data.

Each component is a linear combination of the original predictors - we visualize them as vectors in the feature space.



The Intuition Behind PCA (cont.)

Transforming our observed data means projecting our dataset onto the space defined by the top *m* PCA components, these components are our new predictors.



The Math behind PCA

PCA is a well-known result from linear algebra. Let **Z** be the n x p matrix consisting of columns $Z_1,\ldots, Z_p$ (the resulting PCA vectors), **X** be the n x p matrix of $X_1,\ldots, X_p$ of the original data variables (each standardized to have mean zero and variance one, and without the intercept), and let **W** be the p x p matrix whose columns are the eigenvectors of the square matrix $\mathbf{X}^T\mathbf{X}$, then:

$$\mathbf{Z}_{n\times p} = \mathbf{X}_{n\times p}\mathbf{W}_{p\times p}$$

Implementation of PCA using linear algebra

To implement PCA yourself using this linear algebra result, you can perform the following steps:

- Standardize each of your predictors (so they each have mean = 0, var = 1).
- Calculate the eigenvectors of the $\mathbf{X}^T\mathbf{X}$ matrix and create the matrix with those columns, **W**, in order from largest to smallest eigenvalue.
- Use matrix multiplication to determine $\mathbf{Z} = \mathbf{XW}$.

Note: this is not efficient from a computational perspective. This can be sped up using Cholesky decomposition.

However, PCA is easy to perform in Python using the `decomposition.PCA` function in the `sklearn` package.

PCA example in sklearn

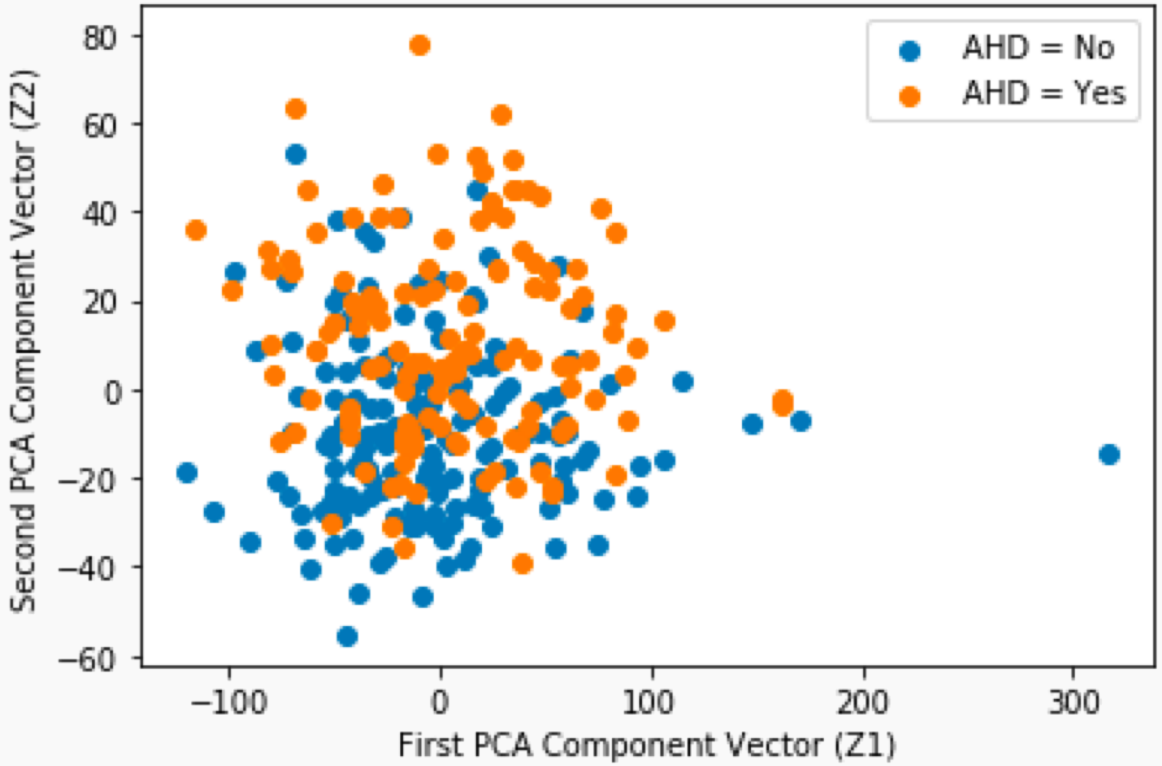
In [11]:
```
X = heart_df[['Age','RestBP','Chol','MaxHR']]

# create/fit the 'full' pca transformation
pca = PCA().fit(X)

# apply the pca transformation to the full predictor set
pcaX = pca.transform(X)

# convert to a data frame
pcaX_df = pd.DataFrame(pcaX, columns=[['PCA1' , 'PCA2', 'PCA3', 'PCA4']])

# here are the weighting (eigen-vectors) of the variables (first 2 at least)
print("First PCA Component (w1):",pca.components_[0,:])
print("Second PCA Component (w2):",pca.components_[1,:])

# here is the variance explained:
print("Variance explained by each component:",pca.explained_variance_ratio_)
```

```
First PCA Component (w1): [ 0.03839966  0.05046168  0.99798051 -0.0037393 ]
Second PCA Component (w2): [ 0.180616    0.10481151 -0.01591307 -0.9778237 ]
Variance explained by each component: [0.74831735 0.15023974 0.0852975  0.01614541]
```

PCA example in sklearn

A common plot is to look at the scatterplot of the first two principal components, shown below for the Heart data:

What do you notice?



PCA for Regression (PCR)

PCA for Regression (PCR)

PCA is easy to use in Python, so how do we then use it for regression modeling in a real-life problem?

If we use all p of the new Z_j, then we have not improved the dimensionality. Instead, we select the first M PCA variables, $Z_1,...,Z_M$, to use as predictors in a regression model.

The choice of M is important and can vary from application to application. It depends on various things, like how collinear the predictors are, how truly related they are to the response, etc...

What would be the best way to check for a specified problem?

Cross Validation!!!

A few notes on using PCA

- PCA is an unsupervised algorithm. Meaning? It is done independent of the outcome variable.
    - Note: the component vectors as predictors might not be ordered from best to worst!
- PCA is not so good because:
    1. Direct Interpretation of coefficients in PCR is completely lost. So do not do if interpretation is important.
    2. Will often not improve predictive ability of a model.
- PCA is great for:
    1. Reducing dimensionality in high dimensional settings.
    2. Visualizing how predictive your features can be of your response, especially in the classification setting.
    3. Reducing multicollinearity, and thus may improve the computational time of fitting models.

Interpreting the Results of PCR

- A PCR can be interpreted in terms of original predictors…very carefully.
- Each estimated β coefficient in the PCR can be *distributed* across the predictors via the associated component vector, *w*. An example is worth a thousand words:

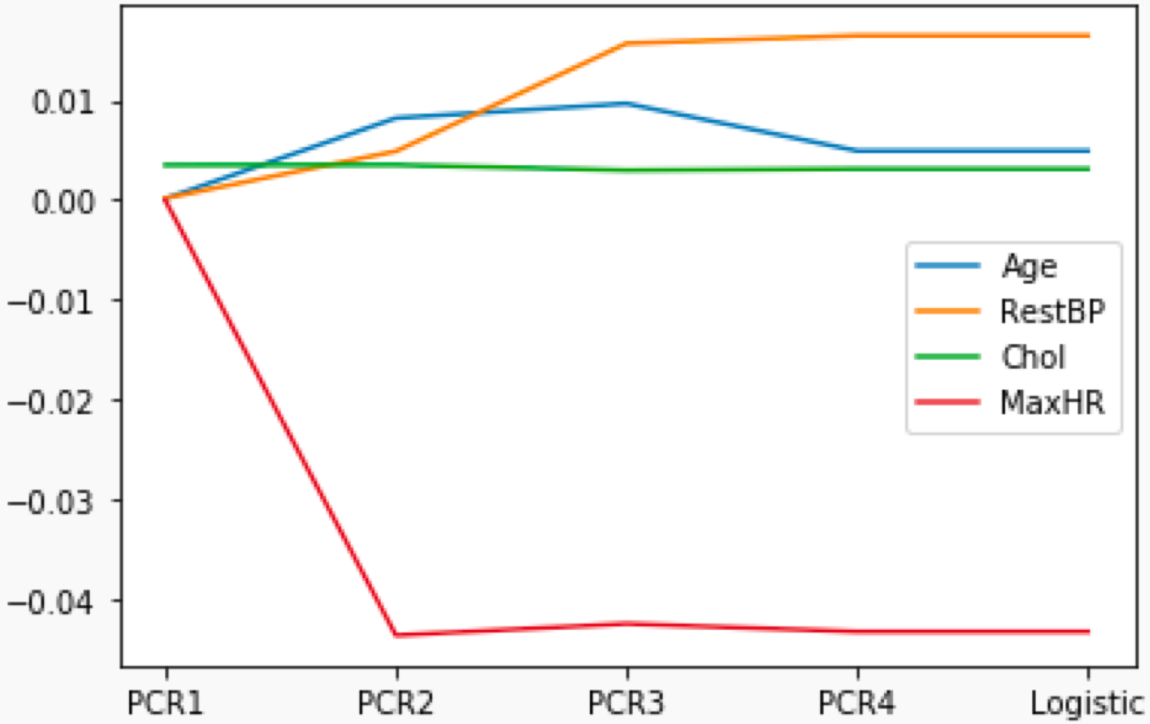
```
In [27]: logit_pcr1 = LogisticRegression(C=1000000,solver="lbfgs").fit(pcaX_df[['PCA1']],y)

         print("Intercept from simple PCR-Logistic:",logit_pca1.intercept_)
         print("'Slope' from simple PCR-Logistic:", logit_pca1.coef_)

         print("First PCA Component (w1):",pca.components_[0,:])


         Intercept from simple PCR-Logistic: [-0.1662098]
         'Slope' from simple PCR-Logistic: [[0.00351092]]
         First PCA Component (w1): [ 0.03839966  0.05046168  0.99798051 -0.0037393 ]
```

- So how can this be transformed back to the original variables?

$$\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 Z_1 = \hat{\beta}_0 + \hat{\beta}_1(\vec{w}_1^T \mathbf{X}) = \hat{\beta}_0 + \hat{\beta}_1 \vec{w}_1^T(\mathbf{X})$$
$$= -0.1162 + 0.00351 \cdot (0.0384X_1 + 0.0505X_2 + 0.998X_3 - 0.0037X_4)$$
$$= -0.1162 + 0.000135(X_1) + 0.000177(X_2) + 0.0035(X_3) - 0.000013(X_4)$$

As more components enter the PCR…

- This algorithm can be continued by adding in more and more components into the PCR.



- This plot was based on a PCA with non-standardized predictors. How would it look different if they were all standardized (both in PCA and plotted here in standardized form of X)?

The same plot with standardized predictors in PCA