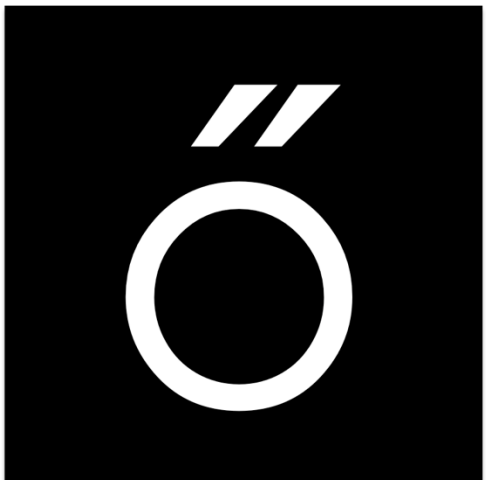
The Erdős Institute



2021 Fall Data Science Program

Lesson Contents

1. A Broad Overview

`Lecture Notebooks/Introduction/2. A Broad Overview.ipynb`

Focused on handling two kinds of data problems – supervised and unsupervised learning

Supervised Learning: Given a set of labels *y* and a set of features *X* (also called predictors or input data), build an algorithm/model that will predict *y* for a set of *X*

- Ex: Predict if a stock's price will increase based on its pre-trade characteristics

Unsupervised Learning: Using a set of features *X*, with no corresponding label, learn something "meaningful" about the data

- Ex: Identifying customer types based on their purchasing habits
- We'll be getting to this later in the program, focusing first on supervised learning

2. Supervised Learning Framework

`Lecture Notebooks/Supervised Learning/1. Supervised Learning Framework.ipynb`

The framework: There is some variable *y* that we are interested in predicting/explaining and a collection of *m* features stored in an m-dimensional variable X. We are going to do our best to identify a statistical model that takes the form $y = f(X) + \epsilon$.

- $f(X)$ is considered the *systematic* information that *X* gives about *y* and can be considered the *signal* that *X* provides about *y*, while ϵ is random noise
- A simple example model would be the linear relationship $y = X + \epsilon$

In practice we cannot precisely obtain f to arbitrary precision due to the presence of random noise/errors, but we can obtain estimates for the model $\hat{f}$ that are quite close

- When fitting a linear relation, "close" might be quantified using low mean square error
- In general one may select different metrics depending on the problem at hand

The two main goals for supervised learning are making predictions and making inferences

When making predictions, the goal is to produce a model/algorithm using training data that can take in new observations and predict an output for them

- The aim here is to make predictions that match the actual values as much as possible

When making inferences, the goal is to produce a model that helps explain the relationship, if any, between *y* and *X*

- Here the aim is to understand how changes in *X* impact *y*
- One example of the "best" estimate in this setting is finding the model that explains as much of the variance in *y* using a minimal number of *X*'s features

Depending on the kind of data encoded in *y*, there are two kinds of supervised learning problems:

Regression problems: These are problems where *y* is a quantitative variable, something counted or measured. Some examples include a person's height or weight, the number of sales made by a company on a given day, and the lifespan of a light bulb.

Classification problems: These are problems where *y* is a qualitative variable, data which are measures of "types" and may be represented by a name, symbol, or a number code. Some examples would be whether a tweet contains misinformation, which digit is represented in an image, or whether an ad buy on a website will result in an app download.

3. Simple Linear Regression

`Lecture Notebooks/Supervised Learning/Regression/1. Simple Linear Regression.ipynb`

We'll start with a simple linear model of the form $y = f(x) + \epsilon = \beta_0 + \beta_1 + \epsilon$, and assume that error ϵ has mean zero and is independent of f

- In order to obtain our estimate $\hat{f}$ we need to estimate $\hat{\beta}_0$ and $\hat{\beta}_1$, which we will do by minimizing a loss function

Mean Square Error: This is the loss function we will minimize to obtain $\hat{f}$ and is given by

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{f}(x_i)\right)^2 = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i\right)^2$$

Using a little bit of calculus, we find that the $\hat{\beta}_0$ and $\hat{\beta}_1$ values that minimize *MSE* are

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

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

Performing linear regression with sklearn

- This is a fairly simple and easy to code model that you could handle any number of ways, but we'll use sklearn because it's a useful tool for a lot of other models we'll get to later
- Here we'll assume we've already imported our data into a pandas dataframe and done a train/test split such that we have a df_train with columns for the *y* and *x* data we want to fit
    - We'll talk about more about train/test splits in the next lesson

Once a model has been obtained, we can compare it to the null model where we simply predict that all *X* values will yield the average value $\bar{y}$

- If the model is a good descriptor of the data, then fit *MSE* should be significantly lower than that of the null model, as is the case in the associated notebook for this lesson

Sample Code:

```
# first need to import the package
from sklearn.linear_model import LinearRegression

# now make the model object
slr = LinearRegression(copy_X=True)
# actually fit the model
slr.fit(df_train['x_data'].values.reshape(-1,1), df_train['y_data'].values)

# get the actual β^_0 and β^_1 estimates
print("beta_0_hat is", slr.intercept_)
print("beta_1_hat is", slr.coef_[0])

# prepare to plot the fit model
x_fit = np.linspace(plot_min,  plot_max)
y_fit = slr.predict(x_fit.reshape(-1,1))
```

Notes on the above code:

- Recall that df_train.y_data and df_train['y_data'] are interchangeable as long as the column label doesn't have a space (in which case you'd need to use the bracket/string method)
- The copy_X=True option is used to make sure that python makes a copy of the data to ensure that it doesn't modify the original dataframe while fitting
    - This is often a non-issue, but it's good to err on the safe side
- The reshape(-1,1) is needed because sklearn expects the *X* input to be a 2D matrix and we're working with 1D data here

4. Train Test Splits

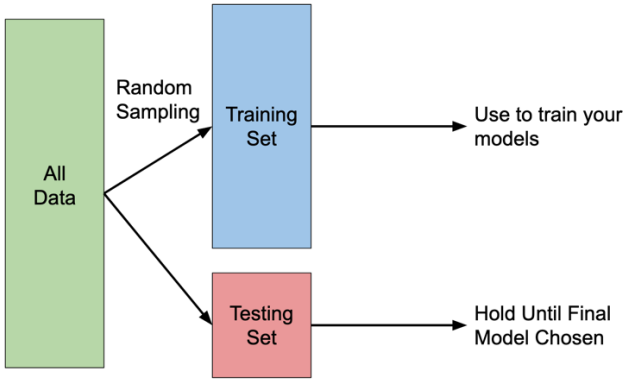
`Lecture Notebooks/Supervised Learning/2. Train Test Splits.ipynb`

For almost all problems we will want to perform a train/test split to help mitigate issues with overfitting and provide a mechanism by which to more robustly validate our modeling results

- Without doing so you may be able to build a seemingly perfect model, but it would likely fail in extraordinary fashion when extrapolated beyond your initial data set

To achieve this, we split our data set into two sets

- The training set encompasses the majority of your data and is what you will use to train your model
- The test set is the smaller of the two sets, the data that you hold out until the end of the model building process as a final check



The primary assumption underlying this technique is that the probability distribution underlying your sample is the same as the distribution underlying the data out in the world

- In theory, performing a random split of the data into a training set and a test set ensures that the distributions are the same for both sets, assuming you have enough observations

Sample Code:

```
# import train_test_split
from sklearn.model_selection import train_test_split


# do the actual split
X_train, X_test, y_train, y_test = train_test_split(X, y, shuffle=True, random_state=604, test_size=0.2)
```

Notes on the above code:

- It's not strictly necessary to set a random state, but it helps to ensure your results are easily reproducible
- The shuffle=True option tells sklearn to randomly shuffle the data before performing the splits, which is generally a good idea
- Using test_size=0.2 means 20% of the data will be reserved for testing
    - Generally between 20% and 30% seems to make for a reasonable test set
- In cases where you're using a categorical variable (explained in detail later) you will usually want to include the stratify option to avoid uneven distributions of your categorical variable between the sets

5. Validation Sets

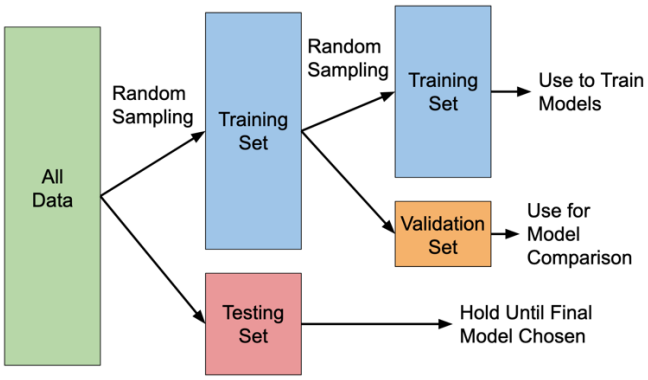
`Lecture Notebooks/Supervised Learning/3. Validation Sets.ipynb`

In predictive modeling we care most about finding models with a low generalization error rather than merely a low training error

- That is, we want to estimate *f* as closely as possible, not merely overfit tiny nuances in our training set that are not representative of the true underlying distribution
- Thus, picking among candidate models solely based on the lowest training MSE is problematic
- Indeed, one cannot measure generalization error using the same data the algorithm was trained on
    - For this we need a validation set (or as we'll see later, multiple validation sets)

Note that we do not want to use our initial test set for this, since it is meant to be used only as a final check once we've chosen a model

We can obtain a validation set by further splitting our training set as shown in the image below



In practice, we can make this split using sklearn's train_test_split the same as for the initial train/test split

- Here again, setting aside between 20% and 30% for the validation set is generally reasonable

6. Cross-Validation

`Lecture Notebooks/Supervised Learning/4. Validation Sets.ipynb`

Recall that our goal in predictive modeling is to make good predictions on new data

- That is, data that our model was not trained on
- Error on new data is often referred to as generalization error because it measures how well our model generalizes beyond the training set

Let's consider the error on a new draw of data to be a random variable that we'll call G

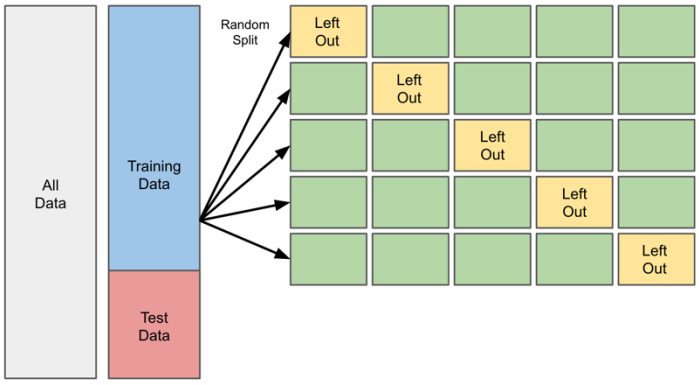
- A validation set in essence provides us with a single observation of G, but what we'd really like is to have a lot of observations so we can infer something about the distribution from which G is drawn

To accomplish this, we'll need to leverage the law of large numbers

- For a sequence $X_1, X_2, \dots, X_n$ of n independent identically distributed random variables with true mean μ, the law of large numbers says that $\lim_{n\to\infty} \bar{X} = \mu$
- Essentially, the mean of a set of random draws will approximately equal the true mean of the distribution from which they are drawn
- Thus, if we can generate a sequence of error observations $G_1, G_2, \dots, G_n$ then we know that $\bar{G} \approx E(G)$

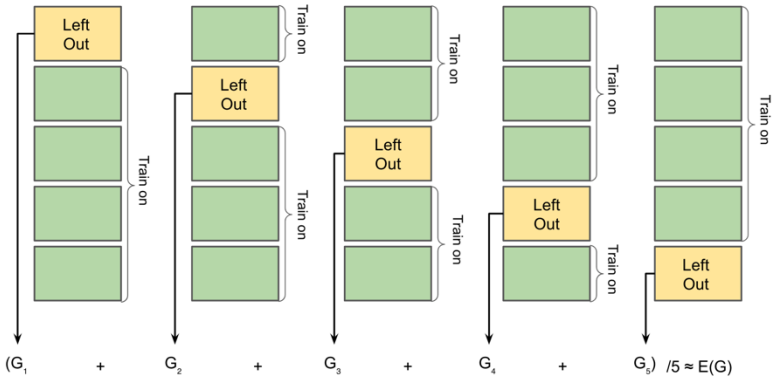
In order to get this sequence of observations, we use k-fold cross-validation

- Split the training data into k equal (or at least roughly equal) chunks



Then generate "observations" of G by cycling through each of the k chunks

- Train your model on the $k-1$ other chunks and then calculate the error on the left out set
- At the end you will have k observations of G, and the mean of $G_1, \dots, G_k$ will give you an estimate of $E(G)$



In general, cross-validation is preferable to using a validation set since all else being equal it is better to have a group of observations of the generalization error rather than just a single observation

- However practical constraints often dictate that you use a validation set instead

- Common reasons for using a validation set include having a relatively small amount of data to use for your analysis or working with models that take considerable amounts of time to train

You could code up *k*-fold cross-validation splits in a number of ways, but we'll focus on using sklearn here

Sample Code:

```
# import KFold
from sklearn.model_selection import KFold

# make a KFold object
kfold = KFold(n_splits=5, shuffle = True, random_state=614)

# when fitting a model we'd do something like the following
for train_index, test_index in kfold.split(X_train, y_train):
    # get the kfold training data
    X_train_train = X_train[train_index,:]
    y_train_train = y_train[train_index]

    # get the holdout data
    X_holdout = X_train[test_index,:]
    y_holdout = y_train[test_index]

    # then fit your model
    # and record the error on the holdout set
```

Notes on the above code:

- This assumes you've already made a standard train/test split to begin with
- More practical applications of this will follow shortly

7. Multiple Linear Regression

`Lecture Notebooks/Supervised Learning/Regression/2. Multiple Linear Regression.ipynb`

In cases where the variable you are interested in modeling depends on more than one variable you can use multiple linear regression

Here we are going to regress y on a range of variables $X_1, \dots, X_m$ using the following model

$$y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_m X_m + \epsilon = X\beta + \epsilon$$

Note that $X = X_1, \dots, X_m$ and $\beta = \beta_1, \dots, \beta_m$ are just used for convenient vector notation, and observations are denoted as $\left(X^{(i)}, y^{(i)}\right)$ with superscripts to avoid confusion with the features in X

As in simple linear regression, we'll focus on minimizing the *MSE* using our n observations

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - X^{(i)}\hat{\beta}\right)^2$$

Rewriting using some linear algebra (so that here T signifies transpose) we obtain

$$MSE = \frac{1}{n}\left(y - X\hat{\beta}\right)^T(y - X\beta) = \frac{1}{n}\left(y^T y - \hat{\beta}^T X^T y - y^T X\hat{\beta} + \hat{\beta}^T X^T X\hat{\beta}\right)$$

Finding the minimum by taking the derivative with respect to $\hat{\beta}$ and setting it equal to zero we recover the ordinary least squares estimate of the coefficient vector β, an equation sometimes called the normal equation

$$X^T X\hat{\beta} - X^T y = 0 \rightarrow \hat{\beta} = (X^T X)^{-1} X^T y$$

As with simple linear regression, this is a relatively simple prescription, and you could code it up using any number of techniques

- The associated notebook goes over how to do so using numpy/linalg, but I'll just focus on sklearn here

Sample Code:

```
# make some phony data
X_train = np.ones((1000,4))
X_train[:,1:] = np.random.randn(1000, 3)
y_train = 2 + 1*X_train[:,1] - 4*X_train[:,2] + 3*X_train[:,3] + np.random.randn(1000)

# import the LinearRegression object
from sklearn.linear_model import LinearRegression

# make the model object
reg = LinearRegression(copy_X=True, fit_intercept=False)

# fit the model object
reg.fit(X_train, y_train)

# look at coef (this gives β^)
reg.coef_

# make a prediction
y_pred_sklearn = reg.predict(X_train)
```

```
# calculate the mse
print( "The MSE is", np.sum(np.power(y_train-y_pred_sklearn, 2))/len(y_train) )
```

Notes on the above code:

- This assumes you have constructed *X_train* with an initial column of ones
  - You can then set fit_intercept=False and the regression will estimate the intercept as the first entry in reg.coef_
  - Thus reg.coef_ cleanly gives you the entire $\hat{\beta}_0, \ldots, \hat{\beta}_m$ list
- Alternatively, you can leave out this initial column of ones in *X_train* (so that it only contains the actual variables you're regressing on) and set fit_intercept=True
  - In this case reg.intercept_ gives you $\hat{\beta}_0$ while reg.coef gives you the $\hat{\beta}_1, \ldots, \hat{\beta}_m$ list
  - Personally, this seems a bit more intuitive and is probably what I'd use in general
- Note that we do not need to use reshape here

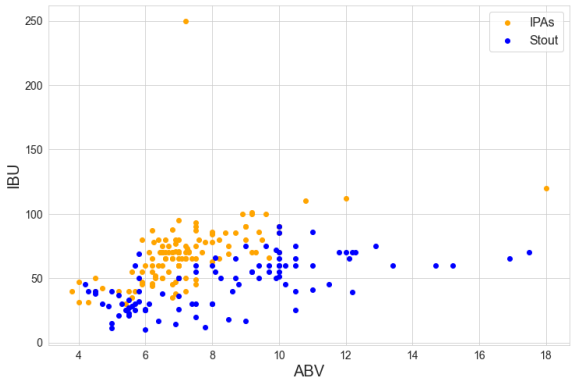
8. Regression: Categorical Variables and Interactions

`Lecture Notebooks/Supervised Learning/Regression/3. Categorical Variables and Interactions.ipynb`

Categorical variables are those that can take on one of a limited number of possible values

- These might include things like book genres, car manufacturers, hair colors, education level, and so on
- In this lesson we'll focus on beer types like IPA and stout

When we examine our data set, it seems that IPAs and stouts behave a little differently, so we'd like to include them in our modeling



In order to do this we'll utilize one-hot encoding to go from human-readable strings to numeric values useable for our regression models

- If you have a variable with *k* unique categories then you will need *k-1* indicator variables to fully one-hot encode the data set

$$I_j = \begin{cases} 1 & \text{if } x = j \\ 0 & \text{if } x \neq j \end{cases}, \quad \text{for } j = 1, \dots, k-1$$

This is quite simple for our two-typed beer data set, as we can one-hot encode it using a single stout indicator

$$I_{Stout} = \begin{cases} 1 & \text{Beer is a Stout} \\ 0 & \text{Otherwise} \end{cases}$$

This would be easy to code up manually, but for future reference we'll look at the get_dummies function that will be very helpful when we progress to more complex problems

Once we have this variable we can do linear regression for a stout-inclusive model of the form

$$IBU = \beta_0 + \beta_1 ABV + \beta_2 I_{Stout} + \epsilon$$

This fits two separate lines, one on the points where $I_{Stout} = 0$ and one on the points where $I_{Stout} = 1$

- However it requires them both to have the same slope β_1
- In order for us to be able to fit the slopes independently we need to include an interaction term

The interaction model is then

$$IBU = \beta_0 + \beta_1 ABV + \beta_2 I_{Stout} + \beta_3 ABV \times I_{Stout} + \epsilon$$

Meaning that

$$\hookrightarrow IBU = \begin{cases} \beta_0 + \beta_1 ABV + \epsilon, & \text{if } I_{Stout} = 0 \\ (\beta_0 + \beta_2) + (\beta_1 + \beta_3) ABV + \epsilon, & \text{if } I_{Stout} = 1 \end{cases}$$

Sample Code:

```
# cleanly import data from a csv into a pandas dataframe
beer = pd.read_csv("~/Erdos/fall-2021/Data/beer.csv")
```

```
# stratify our split on the categorical variable 'Type'
beer_train, beer_test = train_test_split( beer.copy(),
    shuffle=True, random_state=48,
    stratify=df['Type'], test_size=0.2 )

# sample call for get_dummies
pd.get_dummies(beer_train['Beer_Type'])

# store the stout indicator from get_dummies as a new variable
beer_train.loc[:,'Stout'] = pd.get_dummies(beer_train['Beer_Type']).loc[:,'Stout'].copy()

# Make the interaction term
beer_train['ABV_Stout'] = beer_train['ABV'] * beer_train['Stout']
```

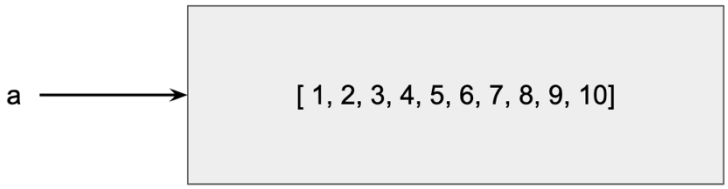
Notes on the above code:

- Here we have included the stratify option to keep the proportions of the 'Type' variable consistent between our train and test sets
    - You'd want to do the same thing if you were making validation sets for this data
- Note that depending on when you make your train/test/validation splits and when you add the indicator and interaction term variables to the dataframe, you may need to manually add them to the test and validation sets
    - If you know in advance that you'll need indicators/interaction terms you can avoid this by creating them in the full dataframe before doing the train/test/validation splits

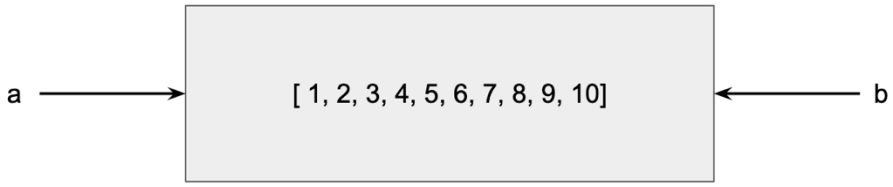
9. Shallow and Deep Copies in Python

`Lecture Notebooks/Python Stuff/1. Shallow and Deep Copies.ipynb`

If you make a list called *a* as a=[1,2,3,4,5,6,7,8,9,10] then python will, not surprisingly, store that list



If you define a new list *b* using b=a, python will point *b* to the same list object that *a* is pointed to



This is called a shallow copy

- If you modify *b* using something like b[4]=11, the same modification will be applied to *a*
- Probably not what you want in most cases

To avoid this problem you need to make a deep copy

- To do this, simply define *b* as b=a.copy()

This is essentially the same thing we're doing when we make train/test splits using df.sample().copy() or when we call LinearRegression(copy_X=True)

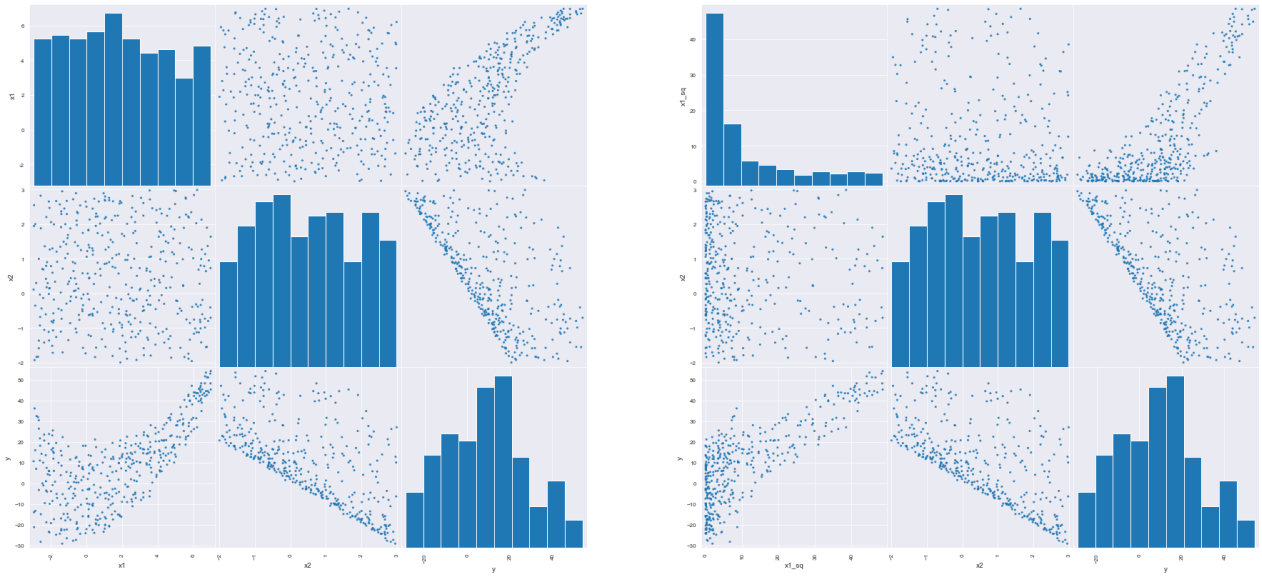
- This will also come up when we implement cross-validation with sklearn's clone method

10. Polynomial Regression and Nonlinear Transformations

`Lecture Notebooks/Supervised Learning/Regression/4. Polynomial Regression and Nonlinear Transformations.ipynb`

When examining data, you may note that certain variables exhibit nonlinear relations with the *y* variable you are interested in modeling

- For instance, x1 in the example plot shown below seems to be quadratic
- Indeed, after transforming to $x1^2$ (labeled 'x1_sq') the relationship with y looks quite linear



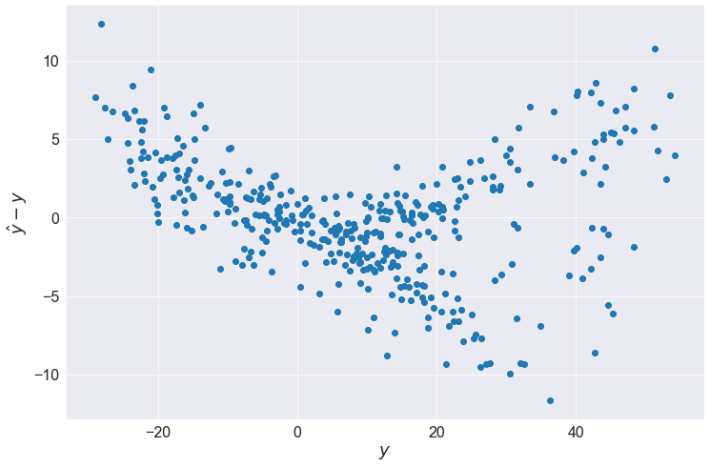
Once you've included the modified term in your dataframe, linear regression can be performed the same as previously

- For the above model we might start by fitting a model of the form $y = \beta_0 + \beta_1 x_1 + \beta_2 x_1^2 + \beta_3 x_2 + \epsilon$
- Note that here the β_2 term will be fit using the x1_sq data

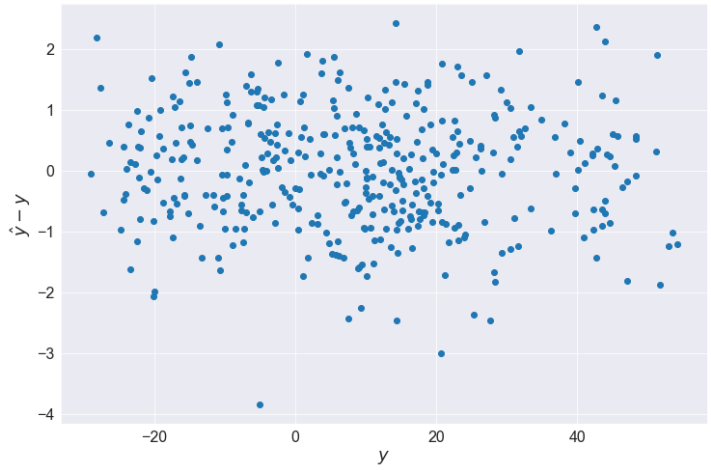
By looking at a plot of $\hat{y} - y$ we can use residuals to help validate our modeling

- When plotting residuals vs *y* for a good model we would expect to see a uniform band of points
- Structure in a residual plot indicates that we're missing some meaningful input in our model
- Here our initial model is clearly flawed, so we might try including an interaction term between x_1 and x_2

$y = \beta_0 + \beta_1 x_1 + \beta_2 x_1^2 + \beta_3 x_2 + \epsilon$



$y = \beta_0 + \beta_1 x_1 + \beta_2 x_1^2 + \beta_3 x_2 + \beta_4 x_1 x_2 + \epsilon$



When analyzing data, oftentimes you just have to do a lot of data exploration since there's no guaranteed way of knowing beforehand whether you'll need an interaction term or not

Note that you generally want to include all lesser powers when modeling a variable

- I.e., if your model includes x_1^3 you should also include x_1^2 and x_1 terms
- Similarly, if you include an interaction term $x_1 \cdot x_2$ you should include both x_1 and x_2 terms

In addition to the polynomial transformations explored here, we could apply the same method to other transformations like √, log, sin, e^x, etc.

Sample Code:

```
# import the scatter_matrix package
from pandas.plotting import scatter_matrix

# plot a scatter matrix to help visualize the data in df_train
scatter_matrix(df_train, figsize=(14,14), alpha=0.9)
plt.show()

# make an additional entry for x1^2
df_train['x1_sq'] = df_train['x1']**2
# also make an x1*x2 entry
df_train['x1x2'] = df_train['x1']*df_train['x2']

# import LinearRegression
from sklearn.linear_model import LinearRegression

# fit the model
reg = LinearRegression(copy_X=True)
reg.fit(df_train[['x1','x1_sq','x2','x1x2']], df_train['y'])
```

Notes on the above code:

- Here scatter_matrix's alpha option determines the opacity of the points to be plotted
  - Smaller alpha → points are more transparent

11. The Bias-Variance Trade-Off

`Lecture Notebooks/Supervised Learning/5. Bias-Variance Trade-Off.ipynb`

High bias typically indicates underfitting the data, while high variance typically indicates overfitting

Recall that our statistical learning framework is built around fitting the model $y = f(X) + \epsilon$ to obtain an estimate of f called $\hat{f}$

As discussed previously, one of the most important characteristics of any given model is its generalization error

- Previously we've used the test or validation set MSEs to estimate this

Letting y_0 and X_0 denote a single test set, we find

$$E[(y_0 - \hat{y}_0)^2] = E\left[\left(y_0 - \hat{f}(X_0)\right)^2\right] = E\left[\left(f(X_0) - \hat{f}(X_0) + \epsilon\right)^2\right]$$

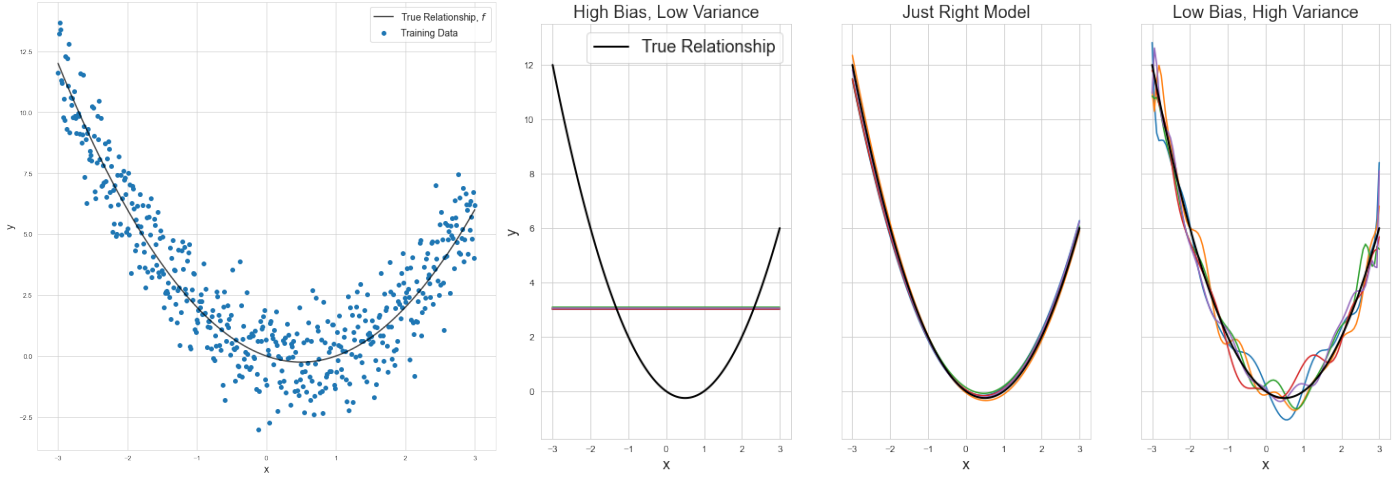
Recalling that $\text{Bias}\left(\hat{f}(X)\right) = E\left(f(X) - \hat{f}(X)\right)$, this can be rewritten as

$$\text{Var}\left(\hat{f}(X_0)\right) + \left[\text{Bias}\left(\hat{f}(X_0)\right)\right]^2 x + \text{Var}(\epsilon) = \text{Variance of } \hat{f} + \text{Bias squared of } \hat{f} + \text{irreducible error}$$

Both Var and Bias^2 are nonnegative, so the best we can do is produce an algorithm with irreducible error $\text{Var}(\epsilon)$

- We can reduce our generalization error by reducing either our model's Bias or its Var
- However, it is often not possible to reduce both simultaneously
- Oftentimes lowering a model's Bias will increase its Var

Consider the set of data shown below, which we will fit iterations of multiple times to explore bias and variance



An example of a model with high bias and low variance for this data set is to take the mean observed y value

- This will significantly underfit the data (which shows a clear pattern) and the bias will be high because the model will be far from the true relationship between y and x
- The variance will be low, however, because with a large enough sample the law of large numbers tells us that the sample mean should be close to $E(y)$
    - Thus s long as our training sample is large enough, then our $\hat{f} = \bar{y}$ model will not vary much over different samples

A linear regression model with low bias but high variance would be one fit using a high degree polynomial of x

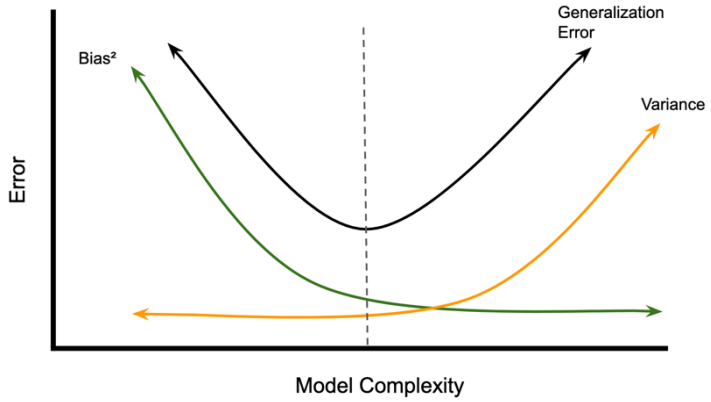
- Here we're fitting a 20 degree polynomial
- The bias here will be low because a high degree polynomial will more closely fit the true relationship

- However the variance will be high because, as the degree of the fitting polynomial increases, the model is increasingly likely to overfit nuances in the training set that do not reflect the underlying probability

Ideally you want to settle on a "just right" Goldilocks model

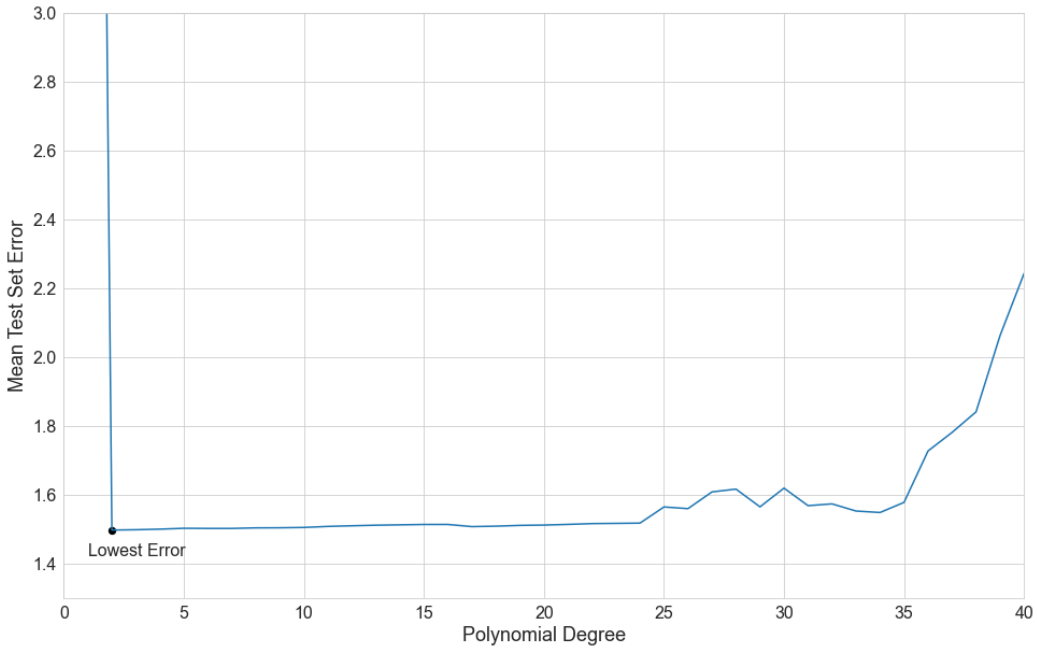
- Looking at the data in this case, that would probably be a quadratic

The model with the lowest generalization error tends to occur somewhere between the extremes of high variance and high bias



Returning again to the sample data set from above and using polynomial degree as our metric for model complexity, we can see this behavior in practice

- At the low end of complexity, where bias dominates in models like $\hat{f} = \bar{y}$, the MSE is huge
- At the high end of complexity with 20+ degree polynomials, variance dominates and MSE increases
- As expected, the lowest MSE is found for a Goldilocks quadratic model



12. Scaling Data

`Lecture Notebooks/Data Preprocessing/2. Scaling Data.ipynb`

Sometimes prior to fitting a model you will need to scale your data, particularly when some features are on vastly different scales than others

There are multiple ways of doing this, but one of the most commonly used ones is to standardize the data like so

$$x_{scaled} = \frac{x - \text{mean}(x)}{\text{standard deviation}(x)} = \frac{x - \bar{x}}{\sigma(x)}$$

This is the transformation applied to turn any arbitrary normal random variable into a standard normal random variable, hence the name standardizing

- This transforms your data to have mean 0 and standard deviation 1
- We could code this up in numpy easily enough, but that'd get tedious quickly
- Instead we'll use sklearn's StandardScaler object, which will also play nicely with train/test splits
- Note that this is not an appropriate scaler to use for one-hot encoded categorical variables

In addition to StandardScaler sklearn offers the MinMaxScaler, which will scale a column of observations such that the values map to the interval [0,1], with the maximum value mapping to 1 and the minimum to 0

Sample Code:

```
# import StandardScaler
from sklearn.preprocessing import StandardScaler

# make a scaler object
scaler = StandardScaler()

# fit the scaler to the training set
scaler.fit(X_train)
# scale the training data, i.e. transform it
X_train_scale = scaler.transform(X_train)

# alternatively, could use
# X_train_scale = scaler.fit_transform(X_train)

# transform the test set
scaler_new.transform(X_test)
```

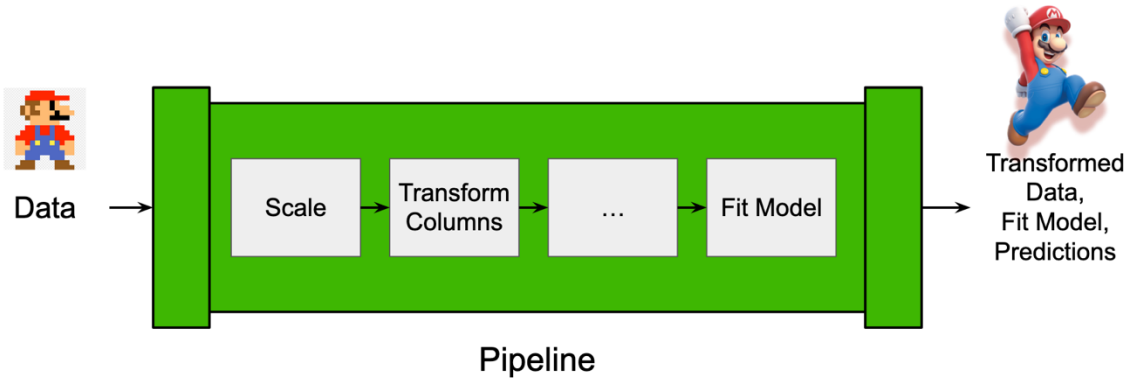
Notes on the above code:

- In the scaler.fit() step StandardScaler goes through your data columns to find/store the mean and standard deviation, which are then applied in the scaler.transform() step
    - Thus, scaler.fit() must be called before scaler.transform()
- You do have the option to do both steps in one by calling scaler.fit_transform(), but you need to be careful about how you choose to do so
    - If you've done all your modeling using a scaled training set, you need to use the training mean and standard deviation when scaling your test/validation sets
    - So you might use scaler.fit_transform() for the training set, but you need to be careful not to refit the scaler when you scale the test data

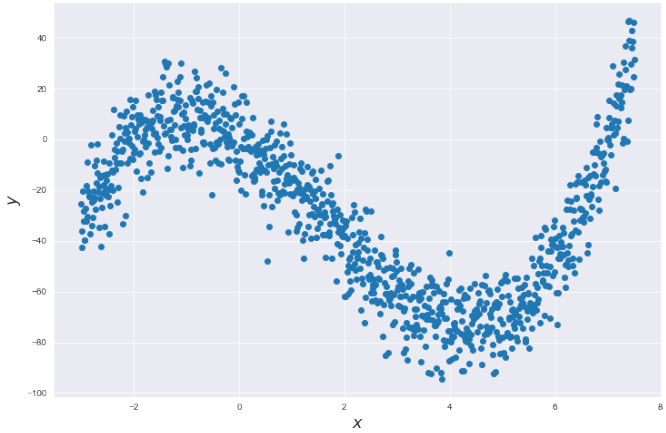
13. Basic Pipelines

`Lecture Notebooks/Data Preprocessing/3. Basic Pipelines.ipynb`

A pipeline is a nice framework for combining the kind of preprocessing steps like scaling, polynomial transforms, and one-hot encoding/interaction terms that we've covered previously into a convenient container that also fits your model, essentially:



Here we will look at using sklearn's PolynomialFeatures and Pipeline functions to analyze this set of toy data



PolynomialFeatures allows us to transform our data cleanly into an array of polynomial transforms

- In the sample code below, for example, we provide a list of x values and this package provides us an array of x, x^2, and x^3 values
  - This can easily be tuned for polynomials of arbitrary degree x^n
- Basically you're automating the process by which we've previously gone through and manually generated columns like x1_sq=x^2

When defining your pipe object using Pipeline, you define your steps as a list

- For the Mario illustration above you'd include scaling, column transforms, the additional steps in …, and then model fitting
- For the simple exercise outlined in the notebook you need only include the PolynomialFeatures transform and then linear regression

Sample Code:

```
# generate toy data
x = np.linspace(-3,7.5,1000)
y = (x-7)*(x+2)*x + 10*np.random.randn(1000)

# import PolynomialFeatures
from sklearn.preprocessing import PolynomialFeatures
```

```
# demonstrate functionality of PolynomialFeatures
poly = PolynomialFeatures(3, interaction_only=False, include_bias=False)
poly.fit_transform(x.reshape(-1,1))

# import Pipeline
from sklearn.pipeline import Pipeline

# make the example Pipeline object
pipe = Pipeline([('poly', PolynomialFeatures(3, interaction_only=False, include_bias=False)),
                 ('reg', LinearRegression(copy_X=True))])

# fit the Pipeline object
pipe.fit(x.reshape(-1,1), y)

# to make predictions
pipe.predict(x.reshape(-1,1))

# to access individual components of Pipeline (in this case the regression coefficients)
pipe['reg'].coef_
# or to recover the polynomial transform output
Pipe['poly'].transform(x.reshape(-1,1))
```

Notes on the above code:

- In this example we've used interaction_only=False because we're only trying to get powers of a single variable x
- We've also used include_bias=False because the linear regression model will handle this for us
    - Note that bias here is referring to the intercept term, so this is related to the discussion in Multiple Linear Regression about incorporating a column of ones in X when fitting
- Here we have again needed to use reshape(-1,1) because x is just a single dimension
- Note that, while we did not need to include a scaler transform for this particular sample pipeline, we could easily have done so by including it in our Pipeline list prior to the polynomial transform

14. Regularization, Ridge, and Lasso Regression

`Lecture Notebooks/Supervised Learning/Regression/5. Regularization Ridge and Lasso Regression.ipynb`

Note that we measure how large a vector β is using a vector norm, denoted as $\|\beta\|$

- When doing ordinary least squares regression, all we were concerned with was minimizing the MSE, but in regularization we constrain ourselves so that we only consider models such that $\|\beta\| < c$
- Basically we try to find the smallest MSE within our constraint budget

In practice, this is equivalent to minimizing the following:

$$\|y - X\beta - \beta_0\|_2^2 + \alpha\|\beta\|, \text{where } \alpha \text{ is some constant and } \|\alpha\|_2^2 = \alpha_1^2 + \alpha_2^2 + \cdots + \alpha_n^2$$

Note that minimizing $\|y - X\beta - \beta_0\|_2^2$ is equivalent to minimizing MSE, so we can think of $\alpha\|\beta\|$ as a penalty term that prevents β from growing too large as we minimize MSE

The amount we "penalize" for a large β depends on the value of α, our first example of a hyperparameter

- A hyperparameter is a parameter we set prior to fitting the model, as opposed to normal parameters like our β coefficients that we estimate during the training process
- For $\alpha = 0$ we recover the standard OLS estimate, while for $\alpha = \infty$ we obtain $\beta = 0$
- Values of α between these extremes will yield different coefficient estimates, and that which gives the best model for your data can be found through cross-validation model comparisons

In ridge regression we take $\|\alpha\|$ to be the square of the Euclidian norm, $\|\alpha\|_2^2$, so we have

$$\|\alpha\|_2^2 = \alpha_1^2 + \alpha_2^2 + \cdots + \alpha_n^2$$

In lasso regression we take $\|\alpha\|$ to be the l_1-norm, so we have

$$\|\alpha\|_1 = |\alpha_1| + |\alpha_2| + \cdots + |\alpha_n|$$

Both of these can be easily implemented using sklearn packages

- Note that it is important to scale your data here, as different scales can have a dramatic impact on how the associated penalty contributions are calculated

Example output from ridge and lasso regression on quadratic data (see sample code below)

Ridge Coefficients

	x^1	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	x^10
alpha=1e-05	-0.882326	-2.302576	-2.475933	11.704747	2.628421	-9.411328	-1.162956	3.047954	0.174927	-0.336191
alpha=0.0001	-0.883752	-2.269963	-2.469638	11.618172	2.620768	-9.329609	-1.159479	3.016233	0.174399	-0.331865
alpha=0.001	-0.897563	-1.971390	-2.408653	10.825525	2.546624	-8.581429	-1.125796	2.725812	0.169278	-0.292263
alpha=0.01	-1.001605	-0.412886	-1.947498	6.685560	1.985160	-4.673622	-0.870541	1.209124	0.130454	-0.085481
alpha=0.1	-1.239782	1.222070	-0.840673	2.298796	0.614306	-0.532465	-0.242007	-0.393939	0.034381	0.132361
alpha=1	-1.214297	1.442458	-0.487714	1.343014	0.008453	0.324187	0.066353	-0.669635	-0.014887	0.161198
alpha=10	-0.851074	0.933740	-0.460300	0.840753	-0.173622	0.366166	0.066685	-0.316627	0.001405	0.058891
alpha=100	-0.306609	0.256087	-0.247117	0.292884	-0.214841	0.266262	-0.145432	0.148309	0.063042	-0.060781
alpha=1000	-0.067605	0.056081	-0.074105	0.085065	-0.088027	0.113003	-0.086398	0.115047	0.008614	-0.016405

Lasso Coefficients

	x^1	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	x^10
alpha=1e-05	-0.888729	-2.261119	-2.446886	11.594987	2.592608	-9.307731	-1.146540	3.007712	0.172416	-0.330698
alpha=0.0001	-0.947788	-1.855304	-2.179150	10.520367	2.262610	-8.293468	-0.995307	2.613748	0.149281	-0.276931
alpha=0.001	-1.358550	0.621964	-0.360308	3.660703	0.060886	-1.662648	0.000000	0.000000	-0.001363	0.083340
alpha=0.01	-1.380527	1.644040	-0.250439	1.369048	-0.000000	-0.000000	-0.000000	-0.466992	0.001718	0.126004
alpha=0.1	-1.239174	1.443382	-0.027485	0.528351	-0.157056	0.000000	-0.000000	-0.000000	0.008652	-0.000949
alpha=1	-0.000000	0.000000	-0.000000	0.000000	-0.125805	0.344885	-0.063567	0.000000	0.000000	-0.000000
alpha=10	-0.000000	0.000000	-0.000000	0.000000	-0.000000	0.000000	-0.000000	0.000000	-0.031708	0.040066
alpha=100	-0.000000	0.000000	-0.000000	0.000000	-0.000000	0.000000	-0.000000	0.000000	-0.000000	0.001077
alpha=1000	-0.000000	0.000000	-0.000000	0.000000	-0.000000	0.000000	-0.000000	0.000000	-0.000000	0.000000

Here we see that the coefficients for lasso regression shrink to 0 quite quickly, while those for ridge regression persist and never quite reach 0

- This is generally the case, and makes feature selection one of the benefits of lasso regression
- Coefficients that tend to stay above 0 as we increase α in lasso regression are typically the most important features for minimizing the training MSE

We can conceptualize why this happens by considering a 2D problem where X and y have means of zero

- In lasso regression we are minimizing $\|y - X\beta\|_1$ under the constraint that $\|\beta\|_1 \leq c$

- For two features, this becomes $|\beta_1| + |\beta_2| \leq c$, which describes a filled square with vertices at (c,0), (0,c), (-c,0), and (0,-c)

- In ridge regression we are minimizing $\|y - X\beta\|_2^2$ under the constraint that $\|\beta\|_2^2 \leq c$
  - For two features, this becomes $\beta_1^2 + \beta_2^2 \leq c$, which is the formula for a filled circle centered at the origin with radius $\sqrt{c}$

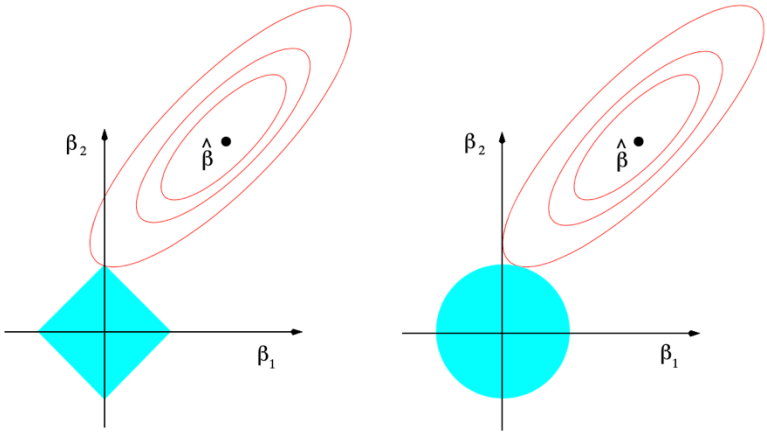


FIGURE 3.11. *Estimation picture for the lasso (left) and ridge regression (right). Shown are contours of the error and constraint functions. The solid blue areas are the constraint regions* $|\beta_1| + |\beta_2| \leq t$ *and* $\beta_1^2 + \beta_2^2 \leq t^2$*, respectively, while the red ellipses are the contours of the least squares error function.*

Given the geometry of the constraint regions, MSE equipotential lines tend to intersect the constraint region at a vertex in lasso regression, while for ridge regression they tend to intersect at a slight offset from the β axes

- In the example above, β_2 is a more significant parameter and we see that in lasso regression intersection occurs at a point where $\beta_1 = 0$, while in ridge regression intersection occurs at a point $\beta_1 < 0$
- We've visualized this in 2D, but the concept generalizes to higher dimensions with $n > 2$ parameters

As a reminder for practical purposes, decreasing α for the sklearn Lasso and Ridge objects increases the size of the constraint region, while increasing α will shrink the constraint region

Pros of lasso regression

- Works well when you have a large number of features that don't have any effect on the target
- Feature selection is a plus, allowing for a sparser model that uses less computational resources

Cons of lasso regression

- Can have trouble with highly correlated features (collinearity), as it typically chooses one variable among those that are correlated, which may be random

Pros of ridge regression

- Works well when the target depends on all or most of the features
- Can handle collinearity better than lasso regression

Cons of ridge regression

- Because ridge regression typically keeps most of the predictors in the model, this can be a computationally costly approach for data sets with a large number of predictors

Elastic net regression fits in between ridge and lasso regression, incorporating both penalty functions and seeking to minimize the function

$$\|y - X\beta - \beta_0\|_2^2 + \overbrace{\alpha_1\|\beta\|_1}^{\text{lasso term}} + \overbrace{\alpha_2\|\beta\|_2^2}^{\text{ridge term}}$$

Sample Code:

```
# re-generate model data (this is the same data we used in the bias-variance tradeoff notebook)
x = np.linspace(-3,3,100)
y = x*(x-1) + 1.2*np.random.randn(100)
```

```
# import necessary packages
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler

# import the two regression models
from sklearn.linear_model import Ridge, Lasso

alpha = [0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]
n=10

# to hold our coefficient estimates
ridge_coefs = np.empty((len(alpha),n))
lasso_coefs = np.empty((len(alpha),n))

# for each alpha value
for i in range(len(alpha)):
  # set up the ridge pipeline
  ridge_pipe = Pipeline([('scale',StandardScaler()),
                ('poly',PolynomialFeatures(n, interaction_only=False, include_bias=False)),
                ('ridge',Ridge(alpha=alpha[i], max_iter=1000000))])

  # set up the lasso pipeline
  lasso_pipe = Pipeline([('scale',StandardScaler()),
                ('poly',PolynomialFeatures(n, interaction_only=False, include_bias=False)),
                ('lasso',Lasso(alpha=alpha[i], max_iter=5000000))])

  # fit the ridge
  ridge_pipe.fit(x.reshape(-1,1), y)
  # fit the lasso
  lasso_pipe.fit(x.reshape(-1,1), y)

  # record the coefficients
  ridge_coefs[i,:] = ridge_pipe['ridge'].coef_
  lasso_coefs[i,:] = lasso_pipe['lasso'].coef_
```

Notes on the above code:

- Here we have upped max_iter from the default value of 1000 because this was not enough to converge
- Note that if you use alpha=0.0 in practice the sklearn regressor object won't work very well
    - If you're interested in this exact value, just use a regular linear regression
    - Alternatively you can usually use something small like alpha=0.000001for most applications

15. Practicing Cross-Validation

`Lecture Notebooks/Supervised Learning/Regression/6. Practicing Cross-Validation.ipynb`

Here we'll work through a sample data set to model car seat sales, fitting a few sample models to demonstrate how to use cross-validation in practice

```
car_train.head()
```

	Sales	CompPrice	Income	Advertising	Population	Price	ShelveLoc	Age	Education	Urban	US
396	6.14	139	23	3	37	120	Medium	55	11	No	Yes
243	7.82	124	25	13	87	110	Medium	57	10	Yes	Yes
367	14.37	95	106	0	256	53	Good	52	17	Yes	No
221	6.43	124	44	0	125	107	Medium	80	11	Yes	No
195	4.19	117	93	4	420	112	Bad	66	11	Yes	Yes

Models to be fit:

$$\text{Sales} = \text{Avg}(\text{Sales}) + \epsilon$$

$$\text{Sales} = \beta_0 + \beta_1 \text{CompPrice} + \epsilon$$

$$\text{Sales} = \beta_0 + \beta_1 \text{CompPrice} + \beta_2 \text{Price} + \epsilon$$

$$\text{Sales} = \beta_0 + \beta_1 \text{CompPrice} + \beta_2 \text{Price} + \beta_3 \text{ShelveLoc\_Bad} + \beta_4 \text{ShelvLoc\_Good} + \epsilon$$

Sample Code:

```
# set the indicator variables
car_train[['ShelveLoc_Bad', 'ShelveLoc_Good']] = pd.get_dummies(car_train['ShelveLoc'])[['Bad', 'Good']]

# import KFold
from sklearn.model_selection import KFold
# import clone
from sklearn.base import clone

# list all potential features for the models
models = [ 'baseline', ['CompPrice'], ['CompPrice', 'Price'],
           ['CompPrice', 'Price', 'ShelveLoc_Bad', 'ShelveLoc_Good'] ]

# list for holding MSEs
cv_mses = np.zeros((5,len(models)))

# loop through cv splits
i = 0
for train_index, test_index in kfold.split(car_train):
  # get train_train data
  car_train_train = car_train.iloc[train_index]
  car_holdout = car_train.iloc[test_index]

  # loop through all models
  j = 0
  for model in models:
    if model == "baseline":
      car_train_train_mean = car_train_train.Sales.mean()
      predict = car_train_train_mean * np.ones(len(car_holdout))
      cv_mses[i,j] = mse(car_holdout.Sales, predict)
```

```
            else:
                reg_clone = clone(reg)
                reg_clone.fit(car_train_train[model], car_train_train.Sales)
                predict = reg_clone.predict(car_holdout[model])
                cv_mses[i,j] = mse(car_holdout.Sales, predict)
            j = j + 1
        i = i + 1

    # get the model with the lowest average cv mse
    models[np.argmin(np.mean(cv_mses, axis=0))]

    # what is the lowest avg cv mse?
    np.mean(cv_mses, axis=0)[np.argmin(np.mean(cv_mses, axis=0))]
```

Notes on the above code:

- This assumes you've already done a train/test split on your data and previously defined an mse function
- The clone object is used to make a deep copy of a base regression object for us to fit each time through our cross-validation loop
    - Always good to err on the safe side and avoid potential issues with shallow copies
- Note that by convention i is usually used as a row counter, while j is typically used as a column counter
    - So in this example i is the counter as we loop through the 5 KFold training/validation sets, while j is the counter as we loop through our possible models
    - Obviously this is arbitrary and you can use other counter variables if you prefer, it's just easy to stick with convention all else being equal

16. Some Regression Model Selection Techniques

`Lecture Notebooks/Supervised Learning/Regression/7. Some Regression Model Selection Techniques.ipynb`

When building models, one of our most important tasks is to identify which features to include in our modeling

- Certainly in most applications you will have some features that do not provide enough predictive power to warrant inclusion in your final model

The most direct, brute force approach is that of best subsets selection, in which you look at every possible model from a set of features

- For each subset of features you train a model and record the CV MSEs, noting at the end which combination of features yields a model with the best (that is, lowest) generalization error
  - Alternatively you could use a single validation set if the data was necessarily limiting
- This can quickly become impractical due to computational constraints, however
  - If you are testing m features you end up needing to fit and assess 2^m models

One approach for avoiding this problem with computational demand is to use "greedy" algorithms that seek to improve the model as much as possible at each individual step of the algorithm

- This generally makes them more efficient, but can come at the cost of overall performance
- We'll look at two approaches: forwards selection and backwards selection

In forwards selection you begin from a baseline model that uses no features and then iteratively improve it by adding individual features

1. Fit the baseline model and record the average CV MSE
2. Fit each of the m possible simple linear regression models and calculate their average CV MSE
   - If none of them outperform the baseline model, you are done
   - Else, choose the one with the lowest average CV MSE to be your new default model
3. Step l: Loop through each of the $m - l$ features not included in the default model, fit the regression model that includes them, and calculate the average CV MSE
   - If none of them outperform the current default model, you are done
   - Else, choose the one with the lowest average CV MSE to be your new default model
4. Repeat Step l until you either find an iteration where the inclusion of new features does not improve upon your default model or you have included all features

Backwards selection essentially reverses this, starting with a model that incorporates all available features and iteratively trimming it down

1. Fit the linear regression model that includes all m features
   - This is your initial default model
2. Fit each of the m linear regression models that result from removing exactly one feature and calculate the resulting average CV MSE
   - If none of them outperform the baseline model, you are done
   - Else, choose the one with the lowest average CV MSE to be your new default model
3. Step l: Loop through each of the $m - (l - 1)$ features still included in the default model, fit the regression model that removes them, and calculate the average CV MSE
   - If none of them outperform the current default model, you are done
   - Else, choose the one with the lowest average CV MSE to be your new default model
4. Repeat Step l until you either find an iteration where the removal of new features does not improve upon your default model or you have removed all features

Note that either of these approaches will terminate in at most $m!$ Steps

Another tool for feature selection is to use lasso regression

- Slowly increase the value of the hyperparameter α and observe the persistence of coefficients
- Coefficients that stay above 0 the longest are likely to be significant
- See associated notebook for a quick example of doing this in practice with the car seats dataset

Sample Code:

```
# this will return a list of all the possible feature combinations  in a list s
def powerset(s):
    power_set = []
    x = len(s)
    for i in range(1 << x):
        power_set.append([s[j] for j in range(x) if (i & (1 << j))])
    return power_set[1:]

# import mse, rather than defining it ourselves
from sklearn.metrics import mean_squared_error

# list of all the models we're going to test using best subsets selection
models = powerset(["CompPrice", "Advertising", "Price", "Population", "ShelveLoc"])

# modify this list to handle our one-hot encoded indicator variables for ShelveLoc
for i in range(len(models)):
    if "ShelveLoc" in models[i]:
        models[i] = [feature for feature in models[i] if feature != "ShelveLoc"]
        models[i].extend(["ShelveLoc_Good", "ShelveLoc_Bad"])

# include the baseline model (which we'll define as taking the average sales in our CV loop)
models.append("baseline")

# mse holder array
cv_mses = np.zeros((5, len(models)))
# base regression object
reg = LinearRegression(copy_X = True)

# loop through all splits
i = 0
for train_index, test_index in kfold.split(car_train):
    # get train and holdout sets
    car_train_train = car_train.iloc[train_index]
    car_holdout = car_train.iloc[test_index]

    # loop through all models
    j = 0
    for model in models:
        if model == "baseline":
    # special handling for the "assume average sales" baseline model
            car_train_train_mean = car_train_train.Sales.mean()
```

```
            predict = car_train_train_mean * np.ones(len(car_holdout))
            cv_mses[i,j] = mean_squared_error (car_holdout.Sales, predict)
        else:
            # clone regression
            reg_clone = clone(reg)
            # fit the regression
            reg_clone.fit(car_train_train[model], car_train_train.Sales)
            predict = reg_clone.predict(car_holdout[model])

            # record mse
            cv_mses[i,j] = mean_squared_error (car_holdout.Sales, predict)
        j=j+1
    i=i+1

# printout the model with the lowest mean mse
print("The model with lowest mean cv mse included the features",
    models[np.argmin(np.mean(cv_mses, axis=0))], "and had an avg cv mse of",
    np.mean(cv_mses, axis=0)[np.argmin(np.mean(cv_mses, axis=0))])
```

Notes on the above code:

- This assumes you've made a 5-fold split using KFold earlier in the script
- You could equivalently write i+=1 (and j+=1) at the end of the CV loops
    - Doesn't really matter beyond i=i+1 being a bit more readable and i+=1 being a bit more efficient

17. Interpreting Regression

`Lecture Notebooks/Supervised Learning/Regression/8. Interpreting Regression.ipynb`

Up to this point we've largely focused on regression as a means of predictive modeling, but here we will instead consider its utility for inferential modeling

- This is a major benefit of regression when compared to some of the more black box-y machine learning algorithms that we'll discuss in later lessons
- Note that we don't need to make train/test splits for this kind of modeling

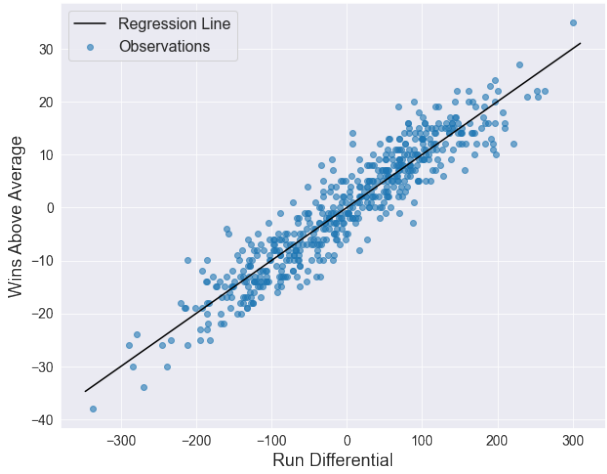
We can use regression models to compare what we expect for y given a value of X

- This holds for either simple or multiple linear regression

Since the model for regressing y on X is $y = X\beta + \epsilon$, for a given value X_{test} the expected value of y is

$$E(y|X = X_{test}) = E(X_{test}\beta) + E(\epsilon) = X_{test}\beta$$

One interesting data set we can look at for inferential modeling is that of wins (here transformed slightly into wins above average) vs run differential in baseball



We can then use our linear regression to make inferences

- For example, from the above data we can infer that a team with a +10 run differential should average about 1.0 wins above average (corresponding to about 82 wins in a season)

It is straightforward to interpret the values of our $\hat{\beta}$ coefficients here

- $\hat{\beta}_1$ is the slope of the plotted line, so increasing run differential by one should yield $\hat{\beta}_1$ additional wins
  - In this case it turns out that increasing RD by 1 is expected to yield about 0.1 more wins
- Similarly, in order to increase our win total by one we expect that we'd need to increase our run differential by $1/\hat{\beta}_1$
  - So we need to increase RD by about 10 to yield an additional expected win
- $\hat{\beta}_0$ is the intercept, telling us the wins above average expected for a team with a +0 run differential
  - Unsurprisingly, a team with a +0 run differential is expected to average 0 wins above average

This can be extended to changes in individual predictors in a multiple regression scenario fairly easily

Consider again the car seats data, to which we've fit the following model:

$$\text{Sales} = \beta_0 + \beta_1\text{CompPrice} + \beta_2\text{Price} + \beta_3\text{Advertising} + \beta_4\text{ShelveLoc\_Good} + \beta_5\text{ShelveLoc\_Bad}$$

After fitting this model, we'd be able to say that a 10 unit change in Price, while all other variables are held constant, would be expected to yield $10\hat{\beta}_2$ additional sales

- You could also examine how changes in multiple continuous features changes the expected sales
    - Like, for example, if you wanted to look at a 10 unit increase in Price combined with a 4 unit change increase in Advertising

A note on the intercept term β_0

- Because we've used categorial variables to encode the shelve location, the third possibility of a "medium" location is not explicitly included in the model
    - Recall that for k possible values we need only encode $k-1$ indicator variables
- In practice that means that β_0 is the intercept for items at a "medium" shelf location, which can be seen by looking at how the Sales model simplifies when all parameters are taken to be zero

$$\beta_0 + \beta_1 \cdot 0 + \beta_2 \cdot 0 + \beta_3 \cdot 0 + \beta_4 \cdot 0 + \beta_5 \cdot 0 = \beta_0$$

The intercept terms for "bad" and "good' locations can be found in a similar fashion

- For ShelveLoc_Bad=1, the intercept is $\beta_0 + \beta_1 \cdot 0 + \beta_2 \cdot 0 + \beta_3 \cdot 0 + \beta_4 \cdot 0 + \beta_5 \cdot 1 = \beta_0 + \beta_5$
- For ShelveLoc_Good=1, the intercept is $\beta_0 + \beta_1 \cdot 0 + \beta_2 \cdot 0 + \beta_3 \cdot 0 + \beta_4 \cdot 1 + \beta_5 \cdot 0 = \beta_0 + \beta_4$

We can also explore how we would expect Sales to change if we were to modify the shelf location of a product from "bad" to "good"

- Here all other parameters are held constant so that only shelf location is changed, and we find

$$\text{Sales}_{good} = \overbrace{\beta_0 + \beta_1 \text{CompPrice}_0 + \beta_2 \text{Price}_0 + \beta_3 \text{Advertising}_0}^{\text{Sales}_0} + \beta_4 + \cancel{\beta_5 \cdot 0} = \text{Sales}_0 + \beta_4$$

$$\text{Sales}_{bad} = \beta_0 + \beta_1 \text{CompPrice}_0 + \beta_2 \text{Price}_0 + \beta_3 \text{Advertising}_0 + \cancel{\beta_4 \cdot 0} + \beta_5 = \text{Sales}_0 + \beta_5$$

$$\hookrightarrow \text{Sales}_{good} - \text{Sales}_{bad} = \beta_4 - \beta_5$$

This kind of exercise is extremely helpful for letting you conceptualize how to interpret regressions

- When in doubt, write out the model and its coefficients and do a little bit of algebra!

18. Interval Estimation

`Lecture Notebooks/Supervised Learning/Regression/9. Interval Estimation.ipynb`

Once we've fit our linear regression model $y = X\beta + \epsilon$, we can obtain estimates of y for a given set of features X_0 from $\hat{\beta}$

- Our fit gives us a point estimate for y, $E(y|X = X_0)$, but what are reasonable possible values for the estimate if we were to repeat the data collection and fitting process?
- According to this data, for a given value of X what is the distribution of y

Point estimates are useful, but it is sometimes preferable to have an estimate for the range of possible values, leading to the idea of confidence intervals

- A confidence interval is a range of values that contain the true parameter with some probability
  - This probability is typically given by $100 \cdot (1 - \alpha)$
  - So for the oft-used $\alpha = 0.05$, we can construct a confidence interval (a, b) within which there is a 95% probability that the true parameter lies
  - Note that this is related to the σ significance convention commonly used in physics and astronomy, with $1\sigma \rightarrow \alpha \approx 0.32$, $2\sigma \rightarrow \alpha \approx 0.05$, $3\sigma \rightarrow \alpha \approx 0.003$, and so on
- Note that the randomness in this range is entirely affiliated with our estimation process, as the true parameter is not random at all

When estimating some parameter γ with point estimate $\hat{\gamma}$, the typical construction of a confidence interval is

$$\hat{\gamma} \pm p_{(1-\alpha/2)}\mathrm{SE}(\hat{\gamma}),$$

where $p_{(1-\alpha/2)}$is a probability modifier found by locating the value p_α that satisfies $P(p \leq p_\alpha) = 1 - \alpha/2$ and $\mathrm{SE}(\hat{\gamma})$ is the standard error of the estimate, found by taking the square root of the variance of the estimate

In simple linear regression, the standard error on an estimate of y given features X_0 is

$$\mathrm{SE}(\hat{y}) = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n-2}} \sqrt{\frac{1}{n} + \frac{(X_0 - \bar{X})}{(n-1)s_X^2}}$$

And since the probability modifier is drawn from a studentized *t*-distribution with $n - 2$ degrees of freedom, denoted as $t_{n-2,1-\alpha/2}$, the $(1 - \alpha)$ confidence interval on $\hat{y}$ is given by

$$\hat{y} \pm t_{n-2,1-\alpha/2} \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n-2}} \sqrt{\frac{1}{n} + \frac{(X_0 - \bar{X})}{(n-1)s_X^2}}$$

While we have focused on simple linear regression, you can also derive confidence bounds for multiple linear regression, although they require more complicated derivations

- We won't bother with the derivations here, although should you need them they should be easy to track down with a simple Google search
  - The same is true for prediction intervals (described below) in multiple linear regression
- In practice you'll likely just rely on a package like statsmodels to calculate the intervals

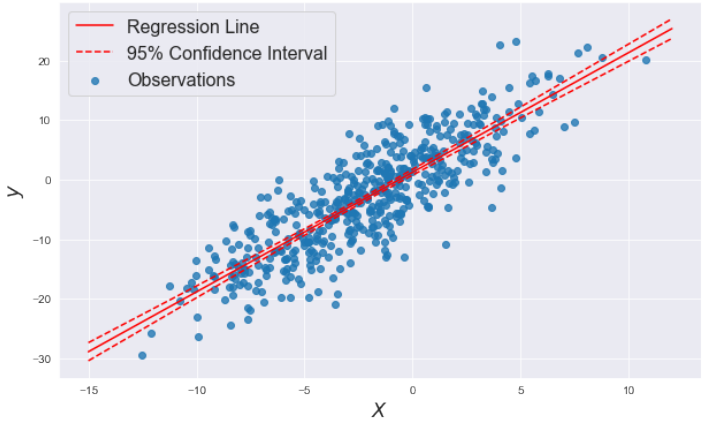
In addition to having a plausible interval for the expected value of y for a given value of X (a confidence interval), it is also useful to have some sense of a plausible interval for actual values of y for a given value of X(a prediction interval)

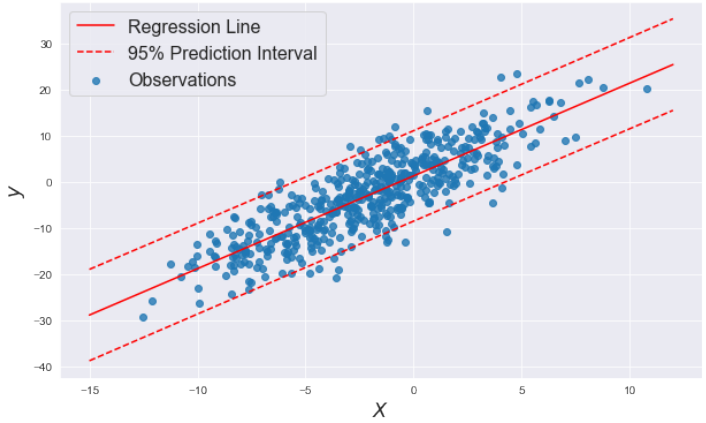
- A confidence interval is an interval for the *average* value of y at a given value of X
- A prediction interval is an interval for *actual* values of y at a given value of X

In simple linear regression the $(1-\alpha)$ prediction interval for $y|X = X_0$ is given by

$$\hat{y} \pm t_{n-2,1-\alpha/2}\sqrt{\frac{\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}{n-2}}\sqrt{1+\frac{1}{n}+\frac{(X_0-\bar{X})}{(n-1)s_X^2}}$$

Note that the prediction interval for a given regression will always be larger than the confidence interval





Sample Code:

```
# make some toy data
np.random.seed(888)
x = 4*np.random.randn(500) - 2
y = 1 + 2*x + 5*np.random.randn(500)

# import t-distribution probability modifier
from scipy.stats import t
# we call this as t.ppf(1-alpha/2, n-2), so a sample call is:
t.ppf(1-.05/2, len(y)-2)

# get the estimate of sigma for epsilon
sigma_hat = np.sqrt(np.sum(np.power(y - slr.predict(x.reshape(-1,1)),2))/(len(y)-2))

# function to get the standard error
def confidence_se(sigma_hat,x_star, x):
  return sigma_hat * np.sqrt((1/len(x) + np.power(x_star - np.mean(x), 2)/((len(x) - 1)*(np.std(x)**2))))

# we'll consider a 95% interval
alpha=0.05

# get confidence interval upper bound
upper = slr.predict(np.linspace(-15, 12, 100).reshape(-1,1)) + t.ppf(1-(alpha/2), len(y)-2)* confidence_se(sigma_hat,
np.linspace(-15, 12, 100), x)
# now the lower bound
lower = slr.predict(np.linspace(-15, 12, 100).reshape(-1,1)) - t.ppf(1-(alpha/2), len(y)-2)* confidence_se(sigma_hat,
np.linspace(-15, 12, 100), x)

# an easy way to make a confidence interval plot in seaborn, skipping the manual definitions above
import seaborn as sns
```

```
sns.regplot(x=x, y=y, ci=95)
plt.show()

#function for the prediction se
def prediction_se(sigma_hat, x_star, x):
  right = np.power(x_star-np.mean(x),2)/((len(x)-1)*np.std(x)**2)
  return sigma_hat*np.sqrt(1 + 1/len(x) + right)

# get prediction interval upper bound
upper = slr.predict(np.linspace(-15, 12, 100).reshape(-1,1)) + t.ppf(1-(alpha/2), len(y)-2)* prediction_se(sigma_hat,
np.linspace(-15, 12, 100), x)
# now the lower bound
lower = slr.predict(np.linspace(-15, 12, 100).reshape(-1,1)) - t.ppf(1-(alpha/2), len(y)-2)* prediction_se(sigma_hat,
np.linspace(-15, 12, 100), x)
```

Notes on the above code:

- This assumes you've previously defined a set of linear toy data y and fit it with a simple LinearRegression object slr

19. Residual Plots

`Lecture Notebooks/Supervised Learning/Regression/10. Residual Plots.ipynb`

If we have a well-fit model $y = X\beta + \epsilon$ and our underlying assumptions are true, when we subtract our estimate from our actual data we should be left with a bunch of random draws from a normal distribution

- If instead our residuals show non-random structure we know there is missing information in our model

Plots of Residuals ($y - \hat{y}$) vs. Features (X)

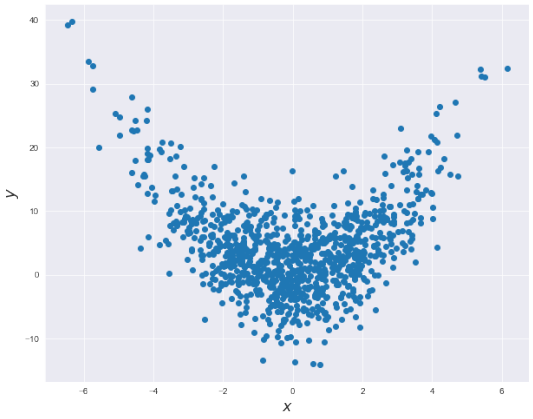
Missed Signal

- These plots are useful for seeing if you are missing some kind of signal in your data, for example by not using a transformation of the feature

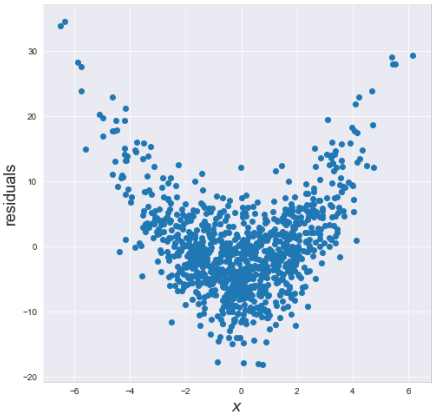
Consider a data set generated by adding a bit of random noise to a quadratic

- If you use a model that only includes a linear term for x, your residuals show clear structure, because your model is missing significant signal that exists in the data set
- If you use an appropriate 2-degree polynomial linear regression your residuals, as expected and as we hope to see, look like a random blob

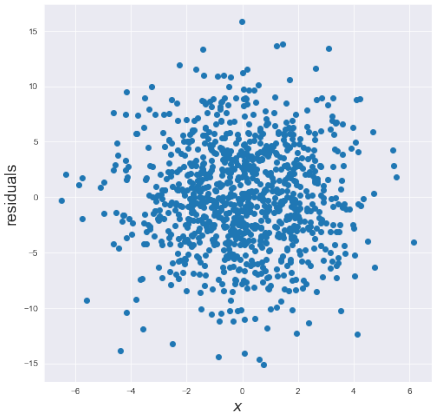
Toy Data



$y = \beta_0 + \beta_1 x$ model

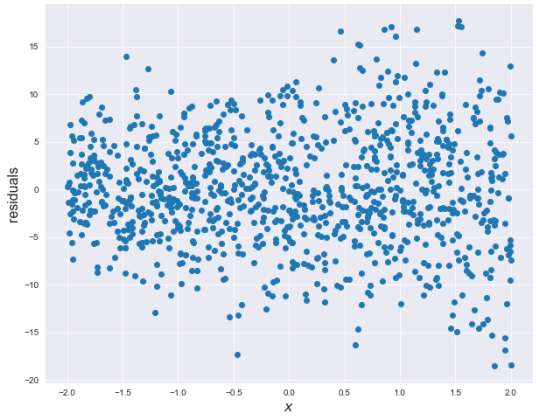


$y = \beta_0 + \beta_1 x + \beta_2 x^2$ model



Equal Variance

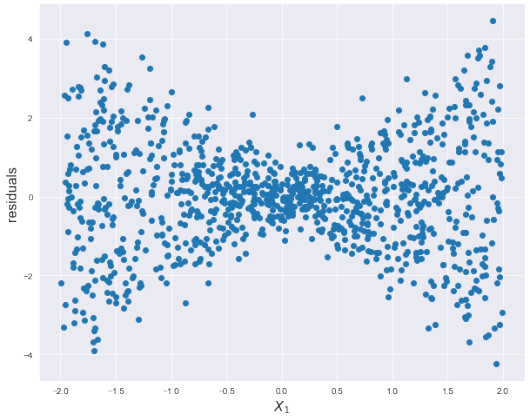
- Another reason we may look at these plots is for assessing our assumption that ϵ is drawn from a uniform distribution for all values of X (an assumption referred to as *homoskedasticity*)
    - A funnel shape (see below) indicates that the variance of ϵ is not equal across all values of X
    - This doesn't usually impact the regression fit too much, but it may impact predictions
    - Note that unequal variances is sometimes referred to as *heteroskedasticity*



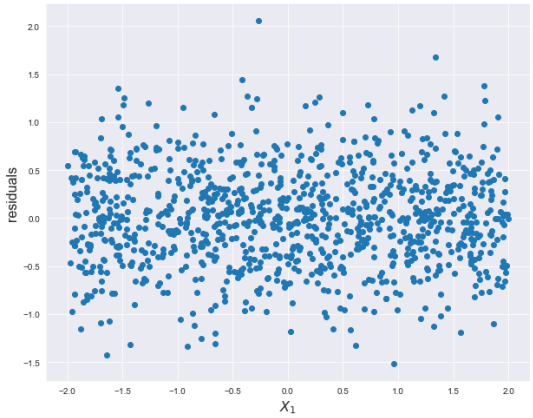
Missed Interactions

- These plots can also be useful for identifying missed interaction terms
  - The presence of a sort of crossed pattern in your residuals suggests a missed interaction
  - For example, if you have data generated from $y = x_1 * x_2 + \epsilon$ and you fit a regression using just $X = (x_1, x_2)$, you have a clear crossed pattern
    - This is rectified when you include the interaction term and use $X = (x_1, x_2, x_1 \cdot x_2)$

$X = (x_1, x_2)$ Regression



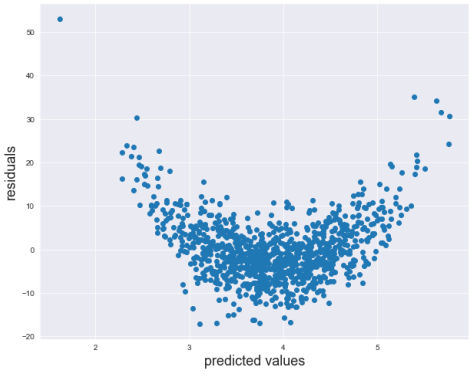
X= $(x_1, x_2, x_1 \cdot x_2)$ Regression



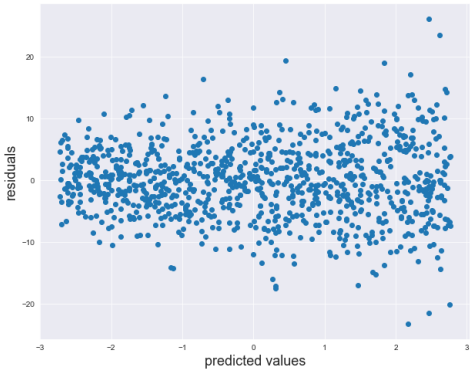
<u>Plots of Residuals $(y - \hat{y})$ vs. Predicted Values $(\hat{y})$</u>

- Often you will want to do this in practice, since plotting individual residual vs features plots can become impractical when you have a large number of features
- These plots can be used to diagnose the same problems discussed above
- See below for how the above examples translate to residuals vs predicted values

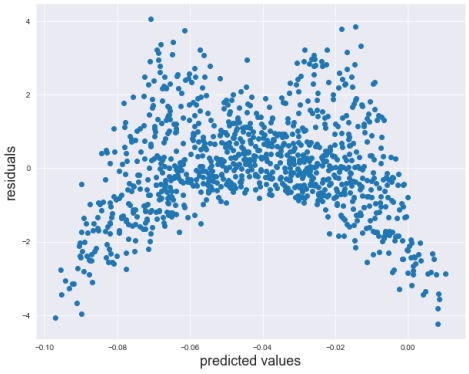
Missed Signal



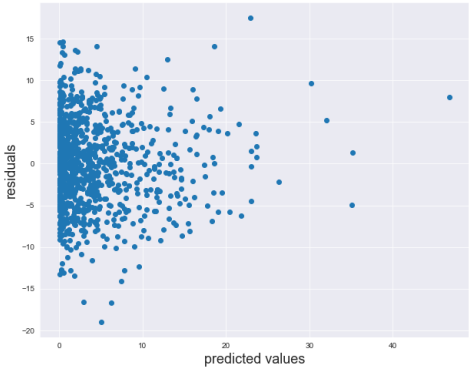
Unequal Variance



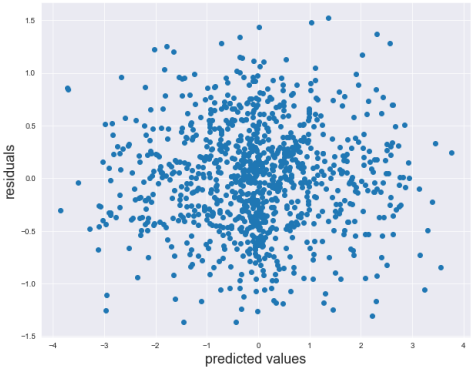
Missed interaction



After including an x^2 term



After including an $x_1 \cdot x_2$ interaction term



20. Weighted Linear Regression

`Lecture Notebooks/Supervised Learning/Regression/11. Weighted Linear Regression.ipynb`

Up to here we've focused on problems where we assume all our data has equal variance, but now we'll look at a regression approach that helps address when this is not the case

Rather than minimizing MSE, we'll look at minimizing a weighted MSE for a set of chosen weights $w^{(i)}$

$$\text{WMSE}\left(\beta, w^{(1)}, \ldots, w^{(n)}\right) = \frac{1}{n}\sum_{i=1}^{n} w^{(i)}\left(y^{(i)} - X^{(i)}\beta\right)^2$$

$$= \frac{1}{n}(y - X\beta)^T W (y - X\beta) = \frac{1}{n}(y^T W y - y^T W X\beta - \beta^T X^T W y + \beta^T X^T W X\beta)$$

Here W is a diagonal matrix with $w^{(i)}$ along the diagonal

Taking the derivative of this with respect to β and setting it equal to zero, we find the following weighted least squares estimate for β:

$$\hat{\beta}_{\text{WLS}} = (X^T W X)^{-1} X^T W y$$

Weighted regression ensures that certain observations are paid greater attention when fitting

- Because those observations with higher weight contribute more to the WMSE, the estimate of $\hat{\beta}$ provided by WLS ensures that those estimates are closer to their actual values
- It is also easy to code up in practice, as all you need to do is give LinearRegression a set of weights to use when fitting the model

When using weighted regression to build a model on data we suspect have unequal variances (or when we know it does because we have measurement uncertainties) we use the following as our weights for each observation

$$w^{(i)} = 1/\sigma_i^2$$

If you are working with data that comes with associated measurement uncertainties, σ_i is generally easy to find

- In physics and astronomy values are typically quoted with 1σ uncertainties, which can be taken as σ_i for the measurement in question
- Thus a quantity 25 ± 2 has $\sigma = 2$ and an associated weight $w = 1/2^2 = 0.25$
- Note that uncertainties reported using 95% confidence intervals are approximately equivalent to being quoted with 2σ uncertainties

More generally, we can estimate σ_i from our data set, and one means of doing so is to:

1. Fit the typical linear regression model for the model you are interested in
2. Calculate the residuals for that model
3. Regress the absolute value of the residuals for that model on the predicted values
    - Call this regression the w regression for reference
4. Use the w regression model to get the σ_i values, and then use these to obtain the weights $w^{(i)}$
5. Finally, use these weights to fit your weighted linear regression model

Sample Code:

```
# need to import LinearRegression
from sklearn.linear_model import LinearRegression


# get the variances
sigma_is = w_reg.predict(predicted.reshape(-1,1))
```

```
# get the weights
w_is = 1/np.power(sigma_is, 2)

# fit the weighted linear regression model
wls = LinearRegression(copy_X=True)
wls.fit(x.reshape(-1,1), y, sample_weight=w_is)
```

Notes on the above code:

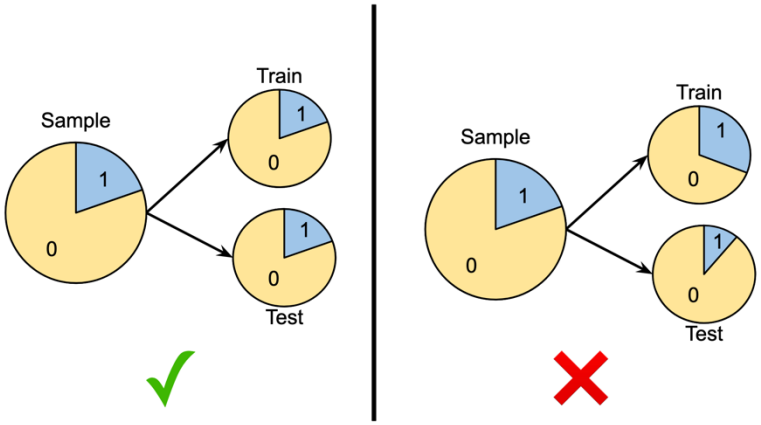
- This assumes you've already fit a standard linear regression
- A full worked example of the above method can be found in the associated notebook for this lesson

21. Train/Test Splits for Classification

`Lecture Notebooks/Supervised Learning/Classification/1. Train Test Splits for Classification.ipynb`

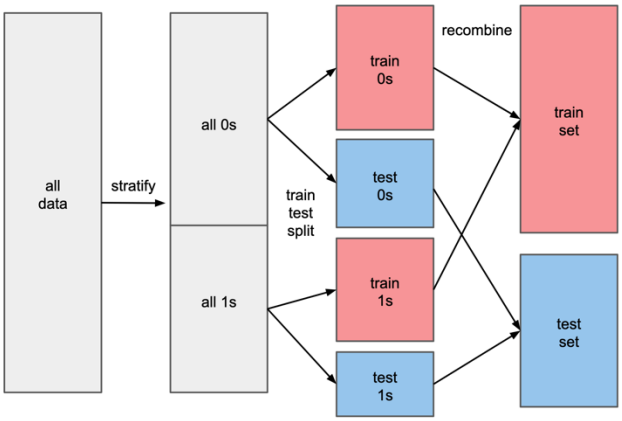
A major assumption in supervised learning is that your data is always being drawn from the same underlying probability distribution

- So when we make any kind of data split we want both sets in the split to look approximately the same
- This is particularly important when your data is significantly imbalanced, meaning one of the categories occurs more frequently than the other(s)



We can ensure that our splits are representative of the sample's distribution by using stratification

- When we perform a data split stratified on a categorical variable we break our sample into the observations corresponding to each unique category
- We then perform a randomized split on each of those subsets
- After the random split, all of the respective categories are recombined into two unique data sets with categorical splits roughly equal to the original sample distribution



This is very easy to implement in practice

- For a single train/test split using sklearn's train_test_split function, just include a stratify argument
- For k-fold cross-validation, use sklearn's StratifiedKFold function

Sample Code:

```
# import train_test_split
from sklearn.model_selection import train_test_split
# make a stratified train/test split
beer_train, beer_test = train_test_split( beer.copy(), shuffle=True,  random_state=48, stratify=beer['Beer_Type'] )
```

```
# import StratifiedKFold
from sklearn.model_selection import StratifiedKFold
# make the kfold object
kfold = StratifiedKFold(n_splits = 5, shuffle=True, random_state=2311)

# loop through train sets and test sets, demonstrating the split on 'Beer_Type'
i = 1
for train_index, test_index in kfold.split(beer_train[['IBU', 'ABV']], beer_train['Beer_Type']):
  # print the beer type splits
  print("Split",i)
  print("CV Training Set Split")
  print(beer_train.iloc[train_index].Beer_Type.value_counts(normalize=True))
  print()
  print("CV Holdout Set Split")
  print(beer_train.iloc[test_index].Beer_Type.value_counts(normalize=True))
  print("++++++++++++++++")
  i = i + 1
```

Notes on the above code:

1. This assumes you've previously imported the beer data set discussed in prior lessons

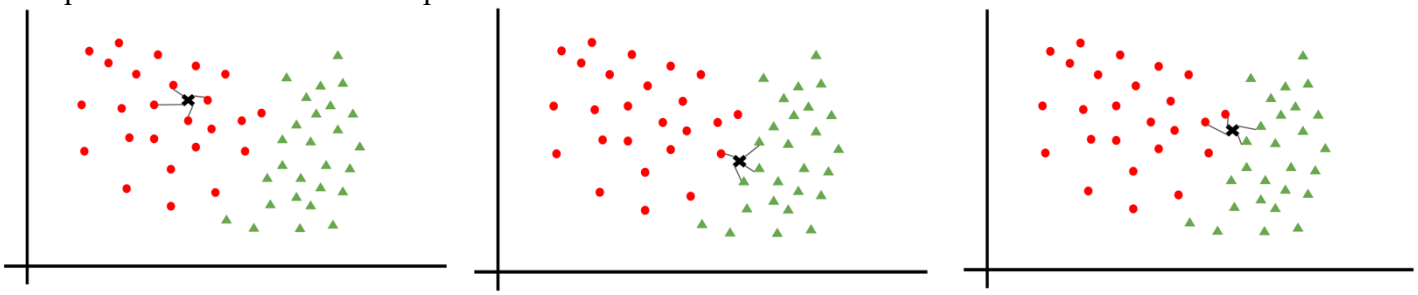
22. *k*-Nearest Neighbors Classifier

`Lecture Notebooks/Supervised Learning/Classification/2. K Nearest Neighbors Classifier.ipynb`

The *k*-nearest neighbors classification algorithm is delightfully straightforward:

1. Input a point you would like to predict on with features X^*
2. Find the k closest points to X^* in the training set, its "nearest neighbors"
    - Note that the k number of points used here is a hyperparameter for the KNN algorithm
3. Tabulate the categories of these nearest neighbors, and whichever category receives the most "votes" is what KNN predicts for point X^*
    - If there is a tie between categories, the prediction is chosen at random from the tied classes

Example KNN classifications for point **x**:



Red Circle | Green Triangle | Randomly Assigned

Note that while we have used Euclidean distance in this simple illustration, we can in principal use any distance metric we'd like

- Also while here we have given each neighbor an equally weighted vote, we could weight the votes
- One scheme for weighing votes is to use the inverse of the distance to our point of interest

We code up KNN using sklearn's KNeighborsClassifier, manually choosing a value for the number of neighbors hyperparameter

- A fun fact to note about KNN is that it isn't actually "fitting" anything per se, since it's really just recording the location of your training set in the given parameter space
- In practice you'll usually want to optimize this with some form of validation/cross-validation test

Note that, because the model does not have a proper training step, the size of the training set does not actually impact the training time for the KNN model

- The time it takes the KNN to make a prediction, however, is significantly impacted by the size of the data set

For many applications it is more useful to have the probability that an observation will be a certain class, rather than the predicted class itself

- We can access this using the predict_proba method
- Note that for unweighted KNN this returns the fraction of nearest neighbors that are of each class

One of the most common ways to judge a classifier's performance is to look at its accuracy, or the proportion of all predictions made that are correct

- This is sort of the general default performance metric for classifiers
- We'll discuss other potentially useful performance metrics in the next lesson

Bias-Variance Tradeoff for KNN:

- When $k = 1$, each point will be assigned the class of the training point to which it is closest
  - Such a model would thus be very sensitive to nuances of the training data and have high variance
- When $k = n$ (where n is the total number of points in the training set) a new data point is predicted using the entire data set
  - Such a model will yield the same prediction for all points, whichever class is most represented in the training set, and thus have minimal variance (and by extension enormous bias)
- In general, lower values of k yield higher variance, while higher values of k yield higher bias

Sample Code:

```
# get the iris data and organize it into a dataframe
from sklearn.datasets import load_iris
iris = load_iris()
iris_df = pd.DataFrame(iris['data'],columns = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width'])
iris_df['iris_class'] = iris['target']

# import KNN package
from sklearn.neighbors import KNeighborsClassifier

# make the model object
knn = KNeighborsClassifier(5)
# fit the model object
knn.fit(iris_train[['sepal_width', 'sepal_length', 'petal_width', 'petal_length']], iris_train['iris_class'])

# predict on the training set
knn.predict(iris_train[['sepal_width', 'sepal_length', 'petal_width', 'petal_length']])

# define an accuracy function
def accuracy(true, predicted):
  return np.sum(true==predicted)/len(predicted)
# get training accuracy for our model
accuracy(iris_train['iris_class'], knn.predict(iris_train[['sepal_width', 'sepal_length', 'petal_width', 'petal_length']]))

# get predict_proba output
knn.predict_proba(iris_train[['sepal_width', 'sepal_length', 'petal_width', 'petal_length']])
```

Notes on the above code:

- This code is built to classify data from the sample iris data set, which provides values for sepal length/width and petal length/width along with classification as one of three different types of irises:
  - (0) setosa, (1) versicolor, or (2) virginica
- Recall the importance of making a stratified train/test split for this kind of analysis
- Here prob_a returns three columns, which contain the probability that, based on votes from the nearest neighbors, points in the training set are either setosa (class 0), versicolor (class 1), or virginica (class 2)
  - The probability in column 0 corresponds to the setosa (class 0) classification, and so on
- We can use distance-weighted votes by including weights='distance' in the KNeighborsClassifier() call

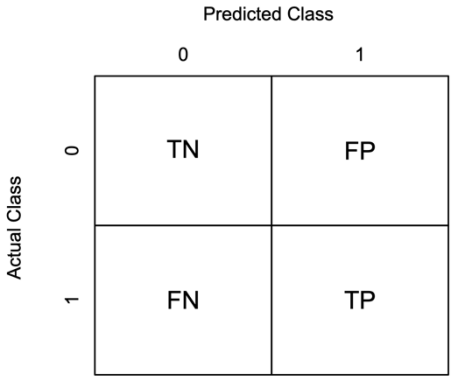
23. The Confusion Matrix, Precision, and Recall

`Lecture Notebooks/Supervised Learning/Classification/3. Confusion Matrix Precision and Recall.ipynb`

Sometimes accuracy is a misleading metric

- For example, if your dataset has an extreme split of, say, 90% in class 0 and 10% in class 1, you could build a classifier with 90% accuracy by simply labeling everything as class 0
- Clearly this is not what we want for many analysis applications

Additional performance metrics can be derived from the confusion matrix, illustrated below for a binary classification problem



The diagonal of the confusion matrix represents data points that are predicted correctly by the algorithm (the "true negatives" and "true positives"), while the off-diagonal represents those that are incorrectly predicted (the "false positives" and "false negatives")

- Note that the confusion matrix can easily be extended for multiclass problems by adding rows and columns, although you lose the precise "false positive"-style nomenclature when doing so

Two popular metrics derived from the confusion matrix are the algorithm's precision and recall:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \qquad \text{and} \qquad \text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Precision asks "out of all points predicted to be class 1, what fraction actually were class 1"

- Essentially, how much should you trust the algorithm when it says something is class 1

Recall asks "out of all the actual data points in class 1, what fraction did the algorithm correctly predict"?

- This estimates the probability that the algorithm correctly detects class 1 data points

We can obtain a confusion matrix for a classifier using sklearn's confusion_matrix

- We could calculate precision and recall from this easily enough, or we can use the precision_score and recall_score functions

We may also be interested in other, somewhat easier to remember, performance metrics; for example:

- Given that an observation is a true positive
  - What is the probability that we correctly predict it is a positive?
    - This is the true positive rate (TPR), and is the same as recall
  - What is the probability that we incorrectly predict it is a negative?
    - This is the false negative rate (FNR)
- Given that an observation is a true negative
  - What is the probability that we correctly predict it is a negative?

        - This is the true negative rate (TNR)
    - What is the probability that we incorrectly predict it is a positive?
        - This is the false positive rate (FPR)

Their associated formulae are:

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \qquad \text{FNR} = \frac{\text{FN}}{\text{TP} + \text{FN}}, \qquad \text{TNR} = \frac{\text{TN}}{\text{TN} + \text{FP}}, \qquad \text{FPR} = \frac{\text{FP}}{\text{TN} + \text{FP}}$$

Depending on the application, we may be interested in optimizing one (or more) of these measures

- For example, if we were trying to predict whether someone has a serious disease we may be most interested in the false negative rate

Sample Code:

```
# import the confusion matrix from sklearn
from sklearn.metrics import confusion_matrix

#display the confusion matrix
confusion_matrix(y_train, y_train_pred)

# record the confusion matrix elements
TN = confusion_matrix(y_train, y_train_pred)[0,0]
FP = confusion_matrix(y_train, y_train_pred)[0,1]
FN = confusion_matrix(y_train, y_train_pred)[1,0]
TP = confusion_matrix(y_train, y_train_pred)[1,1]

# calculate and print recall and precision
print("The training recall is", np.round(TP/(FN+TP),4))
print("The training precision is", np.round(TP/(FP+TP),4))

# import precision and recall
from sklearn.metrics import precision_score, recall_score

# print precision and recall
print("The training recall is", np.round(recall_score(y_train, y_train_pred),4))
print("The training precision is", np.round(precision_score(y_train, y_train_pred),4))

# TPR
print("The training true positive rate is", np.round(TP/(TP+FN),4))
# FNR
print("The training false negative rate is", np.round(FN/(TP+FN),4))
# TNR
print("The training true negative rate is", np.round(TN/(TN+FP),4))
# FPR
print("The training true positive rate is", np.round(FP/(FP+TN),4))
```

Notes on the above code:

- This assumes you've already built a classifier from which to get training set predictions y_train_pred

24. Logistic Regression

`Lecture Notebooks/Supervised Learning/Classification/4. Logistic Regression.ipynb`

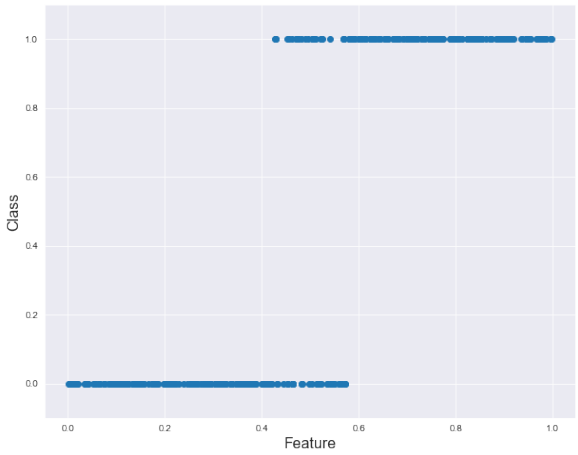
We'll be using logistic regression for binary classification

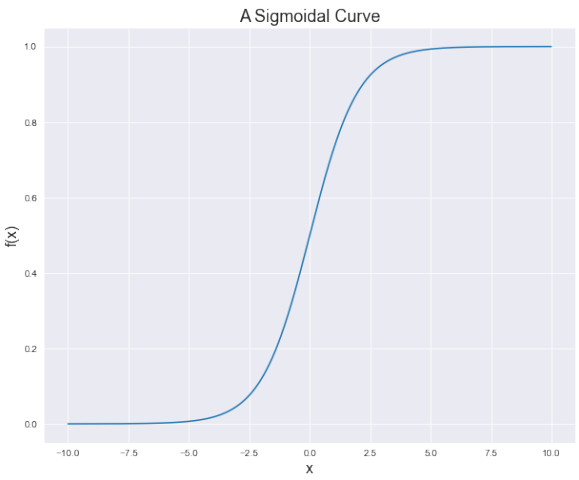
- These are classification problems with only two classes, typically coded as 0 or 1
- Normally the class denoted as 1 is something we want to identify
    - For example, someone that has a disease or someone that qualifies for a loan
- Note that logistic regression can be adapted to multi-class classification problems fairly easily, but we won't explore that here

Logistic regression is a regression algorithm, which means we need to deal with the fact that regression algorithms are used to predict continuous outcomes while binary classification is in no way continuous

- Instead of modeling the output class itself, logistic regression models the probability that a particular data point is an instance of class 1

This is a bit abstract, so let's look at some pictures





While the vertical axis on the above left plot says "class," we could just as easily label it the "probability the instance is 1" since in this case where we know the class of each data point, the probability can be only 0 or 1

Now suppose you have a new data point for which you only have the vector of predictors X

- We're interested in the probability that this point has class $y = 1$, which we'll call $P(y = 1|X) = p(X)$
- A continuous variable, $p(X)$ can take on all values between (and including) 0 and 1

We'll model the probability in logistic regression with a sigmoid curve (above right plot), with the general form

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

This function stays between 0 and 1 and transforms from 0 to 1 in a continuous manner

- Just what we need for regression!

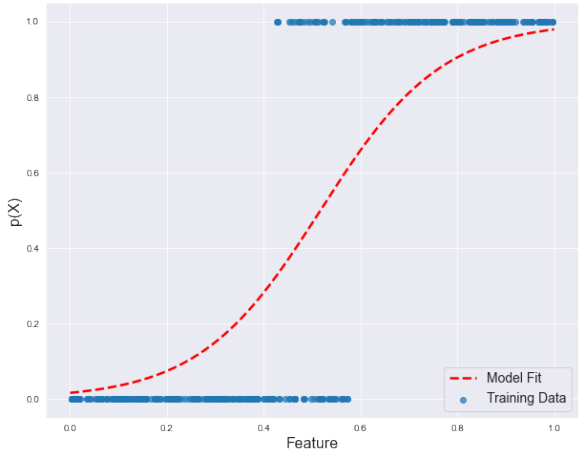
The model that is used in logistic regression is

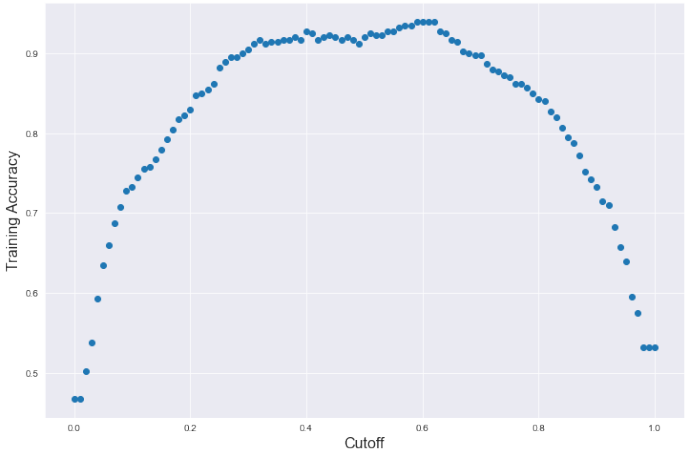
$$p(X) = \frac{1}{1 + e^{-X\beta}}, \text{where}$$

$\beta = (\beta_0, \beta_1, \dots, \beta_m)^T$ is a column of vector coefficients, and X has been extended to include a column of ones

- Rather than using MSE, this model is fit using maximum likelihood estimators

Logistic regression can be implemented in sklearn with LogisticRegression, and after doing so we obtain a fit that looks something like this





The standard approach for generating classifications is to choose a probability cutoff at or above which we classify points as class 1, and below which we classify points as class 0

- A natural choice might be $p(X) \geq 0.5$, but ultimately this is a hyperparameter that you are free to vary
- This is an example of a decision boundary

One particularly nice aspect of this algorithm is that, unlike some others that we'll touch on, we can directly interpret its results

To understand how, we first rearrange our model a bit

$$p(X) = \frac{1}{1 + e^{-X\beta}} \rightarrow \log\left(\frac{p(X)}{1 - p(X)}\right) = X\beta$$

The expression $p(X)/\big(1 - p(X)\big)$ is known as the odds of the event $y = 1$, so the statistical model for logistic regression is really just a linear model for the log odds of being class 1

Consider the simple single-feature (x) model fit in the plot above

$$\log\left(\frac{p(x)}{1 - p(x)}\right) = \beta_0 + \beta_1 x, \text{or } \text{Odds}|x = Ce^{\beta_1}$$

If we increase x from, say, d to $d + 1$, a 1 unit increase, then our odds are e^{β_1} units larger (or smaller, depending on the value of β_1), as demonstrated here:

$$\frac{\text{Odds}|x = d + 1}{\text{Odds}|x = d} = \frac{Ce^{\beta_1(d+1)}}{Ce^{\beta_1 d}} = e^{\beta_1}$$

There are a number of key assumptions made by this model that you will want to validate when working with real-world data:

1. Each sample must be independent from all other samples
2. When using multiple predictors, they shouldn't be correlated
3. The log odds depend linearly on the predictors
4. You should have a fairly large data set to work with

You can also implement regularization with logistic regression (ridge, lasso or elastic net)

- In fact, by default sklearn's LogisticRegression is ridge logistic regression by default

Sample Code:

```
# import LogisticRegression
from sklearn.linear_model import LogisticRegression

# make model object
log_reg = LogisticRegression()
# fit the model
log_reg.fit(X_train.reshape(-1,1), y_train)

# call predict and prob_a
log_reg.predict(X_train.reshape(-1,1))
log_reg.predict_proba(X_train.reshape(-1,1))

# set a cutoff probability for classification
cutoff = .5
# store prediction probabilities
y_prob = log_reg.predict_proba(X_train.reshape(-1,1))[:,1]
# assign classification based on the cutoff
y_train_pred = 1*(y_prob >= cutoff)

# print the accuracy
print("The training accuracy for a cutoff of", cutoff, "is", np.sum(y_train_pred == y_train)/len(y_train))

print("A 0.1 unit increase in our feature multiplies the odds of being classified as 1 by " +
    str(np.round(np.exp(.1*log_reg.coef_[0][0]),2)))
```

Notes on the above code:

- As in prior lessons, we need to include reshape(-1,1) because X_train in this example is one dimensional
- Note that (y_prob >= cutoff) gives us an array of Trues and Falses
    - We then convert it into an array of 1s and 0s when using 1*(y_prob >= cutoff)
- Here prob_a returns two columns, the first of which contains the probability that points are class 0, and the second of which contains the probability that they are class 1

25. The ROC Curve

`Lecture Notebooks/Supervised Learning/Classification/5. ROC Curve.ipynb`

While TPR and FPR are constant for a given classification, as we saw in the previous lesson they can have different classifications if we choose different probability thresholds

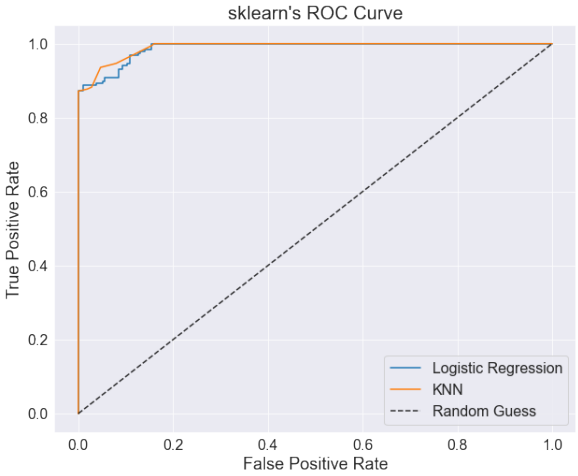
- Choosing $p = 0.4$, for example, will give a different classification from $p = 0.5$

The idea of examining how the TPR and FPR change with different prediction probability thresholds underlies the "receiver operating characteristics curve," or more commonly the ROC curve, which plots TPR against FPR

- Recall that ideally we'd have a TPR of 1 and a FPR of 0

A perfect classifier will have an ROC curve that traces the upper left corner of the unit square, although this will generally not be attainable in practice

- Note that we often compare our ROC curves to the line $y = x$ since this is what a random guess algorithm would produce



We can compare the ROC curves of multiple algorithms by computing the area under the ROC curve, often abbreviated AUC (area under the curve)

- When comparing algorithms, we'd choose that which has the largest AUC score
  - For example, in the above figure the KNN classifier has an AUC of 0.9906, so we would choose it over the Logistic Regression classifier and its 0.9885 AUC score
  - You might make exceptions in practice if computationally expensive algorithms or those with long run times provided only a minor improvement
- We do not need to calculate this ourselves, although we certainly could if we felt so inclined
  - Instead we'll use sklearn's auc, roc_curve, and roc_auc_scoree functions
- Note that the AUC for the random guess algorithm's $y = x$ ROC curve is 0.5

Sample Code:

```
# make simple TPR and FPR functions
def TPR(conf_mat):
  return conf_mat[1,1]/np.sum(conf_mat[1,:])
def FPR(conf_mat):
  return conf_mat[0,1]/np.sum(conf_mat[0,:])

# generate the ROC curve
cutoffs = np.arange(0,1.01,.01)
fprs = []
tprs = []
```

```
for cutoff in cutoffs:
    y_pred = 1*(y_prob >= cutoff)
    fprs.append(FPR(confusion_matrix(y_train, y_pred)))
    tprs.append(TPR(confusion_matrix(y_train, y_pred)))

# plot the ROC curve
plt.figure(figsize=(10,8))
plt.plot(fprs, tprs)
plt.plot(np.linspace(0,1,10), np.linspace(0,1,10), 'k—', alpha=.8, label="Random Guess")

plt.xlabel("False Positive Rate", fontsize=18)
plt.ylabel("True Positive Rate", fontsize=18)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.legend(fontsize=16)
plt.title("The ROC Curve", fontsize=20)
plt.show()

# now demonstrate sklearn's functions, starting by importing them
from sklearn.metrics import auc, roc_curve, roc_auc_score

# use auc to find the area under the curve we found above
auc(fprs, tprs)

# generate an ROC curve with sklearn's roc_curve function
fprs, tprs, thresholds = roc_curve(y_train, y_prob)

# use roc_auc_score to get an AUC score directly
roc_auc_score(y_train, y_prob)

print("The logistic regression model has a",
    np.round(roc_auc_score(y_test, log_reg.predict_proba(X_test.reshape(-1,1))[:,1]),4),
    "AUC on the training data.")
print("The knn model has a",
    np.round(roc_auc_score(y_test, knn.predict_proba(X_test.reshape(-1,1))[:,1]),4),
    "AUC on the training data.")
```

Notes on the above code:

- This assumes you've already done a train/test split and built some classifier which gave you the predictions in y_pred and the probabilities in y_prob
    - The final bit of code assumes that you've gone on to fit a logistic regression and a KNN classifier using roc_curve
        - This is what's shown in the plot above
        - Note that in practice you'd probably want to use CV validation for something like this
- The roc_curve function essentially automates what we did in the cutoff loop
    - Similarly, roc_auc_score gives you the AUC score directly without you having to bother with any of the other steps

26. Bayes' Rule Reminder

`Lecture Notebooks/Supervised Learning/Classification/6. Bayes Rule Reminder.ipynb`

Assume that we have some probability space Ω

Conditional Probability

For events A and B with $P(B) \neq 0$, we define the probability of A conditional on B as

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Law of Total Probability

If $B_1, B_2, \dots, B_n$ are disjoint events such that $\cup_{i=1}^{n} B_i = \Omega$, then for any event A it holds that

$$P(A) = \sum_{i=1}^{n} P(A \cap B_i)$$

Note that what we're saying here – in English – is that we have a set of events B that don't overlap with one another (they're disjoint) and cover the whole space (the $\cup$ condition)

Bayes' Rule

For events A and B with $P(B) \neq 0$, Bayes' rule (or the Bayes-Price theorem) states that

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

"The probability of A occurring given that B occurs is equal to the probability that B occurs given that A occurs times the probability that A occurs divided by the probability that B occurs"

This can then be taken a step further using the law of total probability

$$P(A|B) = \frac{P(B|A)P(A)}{P(B \cap A) + P(B \cap A^c)} = \frac{P(B|A)P(A)}{P(B|A)P(A) + P(B|A^c)P(A^c)}, \text{where } A^c = \Omega - A$$

This is a bit abstract, but we'll put it into practice in our next few lessons

27. Linear Discriminant Analysis (LDA)

`Lecture Notebooks/Supervised Learning/Classification/7. Linear Discriminant Analysis.ipynb`

Although it was originally proposed by Fisher in 1936 as a supervised dimension reduction technique, here we're going to look at linear discriminant analysis as it applies to classification, its more common modern use

Suppose we have a set of m features collected in a matrix X and an output variable y that can take on any of C possible categories

- In the case of logistic regression, where $C = 2$ (that is, y was either class 1 or class 0), we modeled $P(y = 1|X = X^*)$ in order to make predictions

We can use Bayes' rule to rewrite this expression as

$$P(y = 1|X = X^*) = \frac{\pi_c f_c(X^*)}{\sum_{i=1}^{c} \pi_l f_l(X^*)},$$

where π_c denotes the *prior* probability that a random observation comes from the c^{th} class and $f_c \equiv P(X|y = c)$

- This form assumes that X is a qualitative variable, but it can be easily rewritten if X is continuous

When fitting this model we estimate the π_l values by taking the fraction of the sample set belonging to each of the C classes

- So if 1/3 of your sample belongs to class 1, then $\pi_1 = 1/3$, and so on
- We'll need to make some assumptions to estimate the $f_1(X)$ values, however

In linear discriminant analysis we assume that $X|y = c$ is Gaussian

For a single feature (the $m = 1$ case) this means that $f_c(X)$ is

$$f_c(X) = \frac{1}{\sqrt{2\pi}\sigma_c} \exp\left(-\frac{1}{2\sigma_c^2}(X - \mu_c)^2\right),$$

which is the probability density function of a normal random variable with mean μ_c and standard deviation σ_c

- Note that in linear discriminant analysis we assume that $\sigma_1 = \sigma_2 = \cdots = \sigma_c = \sigma$

Under this assumption we find the following estimate for $P(y = c|X)$

$$P(y = c|X) = \frac{\pi_c \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2\sigma_c^2}(X - \mu_c)^2\right)}{\sum_{i=1}^{c} \pi_l \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2\sigma_c^2}(X - \mu_l)^2\right)}$$

We can then estimate μ_c and σ as

$$\hat{\mu}_c = \frac{1}{n_c} \sum_{i:y_i=c} X_i\,, \qquad \hat{\sigma}^2 = \frac{1}{n - C} \sum_{c=1}^{C} \sum_{i:y_i=c} (X_i - \hat{\mu}_c)^2$$

We typically make classifications in this setting by choosing the class, c, for which $P(y = c|X)$ is largest

- This is the $p = 0.5$ cutoff case for a situation with 2 classes like we've looked at previously, it just becomes more complicated here as we expand the number of classes we consider
- Through some algebra and log manipulations you can show that this is equivalent to choosing the class, c, for which the discriminant function is largest

The discriminant function for class c, that we estimate using the $\hat{\mu}_c$ and $\hat{\sigma}^2$ values, is

$$\delta_c = X\frac{\mu_c}{\sigma^2} - \frac{\mu_c^2}{2\sigma^2} + \log(\pi_c)$$

Note that the discriminant is a linear function of X, with a slope μ_c/σ^2 and an intercept $-\mu_c^2/2\sigma^2 + \log(\pi_c)$

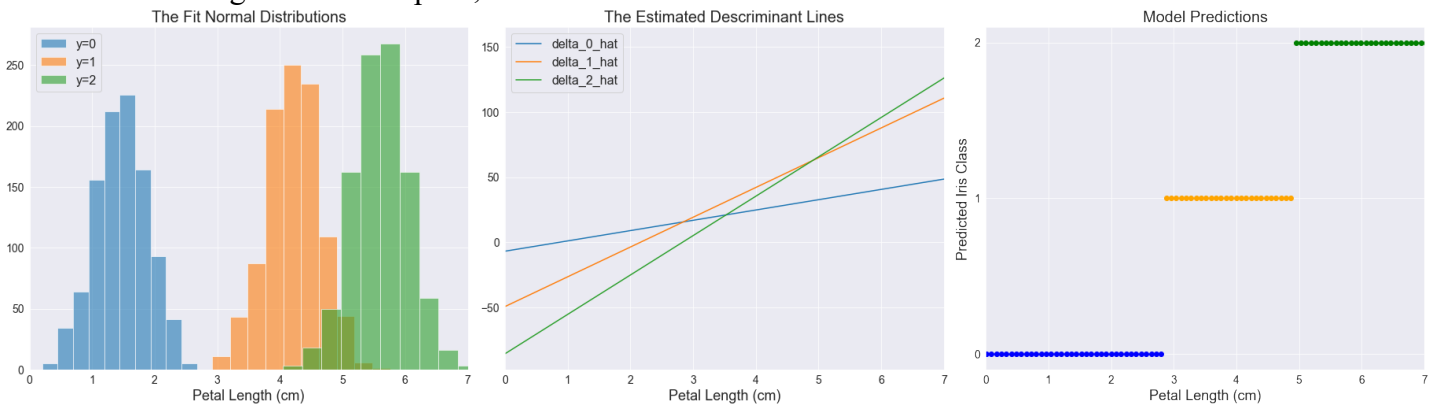
- Hence the name, linear discriminant analysis

In practice you can code up LDA using sklearn's LinearDiscriminantAnalysis function

- The associated notebook also provides a worked example showing how LDA works in practice

The procedure is fairly straightforward (illustrated below using "petal length" to classify iris types):

1. Obtain the π factor for each class by measuring how much of the total sample each class represents
    - In this case the 3 classes are evenly represented, so all are 1/3
2. Measure the mean for each class and calculate the assumed-to-be uniform standard deviation
    - This assumption of uniform variance will be important to validate in practice
3. Calculate the associated δ_c discriminant lines for each class using the formula above
4. Identify the ranges where each class has the largest δ_c, and use these as your predictions
    - I.e., the yellow line is on top from about 3-5 cm, so class 1 would be the predicted class for that range of feature-space, and so on



When we have multiple features we'd like to use when predicting y (the $m > 1$ case), we assume that $f_c(X)$ is a probability density function for a multivariate normal distribution with a class-specific mean vector and a common covariance matrix

- In other words, each column of X is assumed to be a normal distribution whose mean is dependent upon the class you are looking at, and the columns of X also have some correlation with one another that is identical across the possible classes
- This is denoted as $X|y = c \sim N(\mu_c, \Sigma)$, where $\mu_c = E(X|y = c)$ and $\Sigma = \text{cov}(X)$

In this case $f_c(X)$ is given as follows:

$$f_c(X) = \frac{1}{(2\pi)^{m/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(X-\mu_c)^T\Sigma^{-1}(X-\mu_c)\right),$$

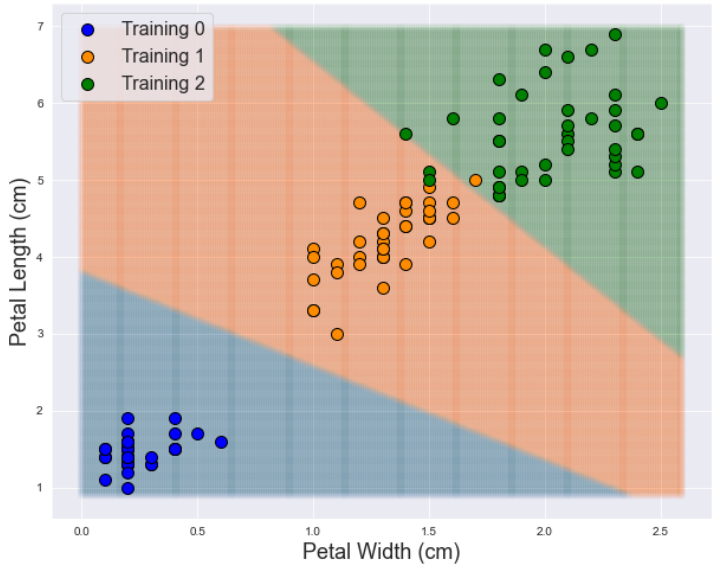
which will result in a class-specific discriminant function of:

$$\delta_c(X) = X^T\Sigma^{-1}\mu_c - \frac{1}{2}\mu_c^T\Sigma^{-1}\mu_c + \log(\pi_c)$$

Estimation of μ_c and Σ are similar to the single feature case, and for any given point in feature-space, X^*, the LDA classifier will select the class, c, with the highest estimated $\delta_c(X^*)$ as its prediction

As an example, if we extend the iris classifier example from using just the "petal length" feature to also use the "petal width" feature, we get the prediction regions shown below using multi-feature LDA classification

- See the associated notebook for the code used to generate the region plot
- The lines that mark the boundary between these regions are an example of decision boundaries



Sample Code:

```
# import LDA package
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# make the model object
LDA = LinearDiscriminantAnalysis()
# fit model object
LDA.fit(X_train.petal_length.values.reshape(-1,1), y_train)

# demonstrate predict_proba
LDA.predict_proba(X_train.petal_length.values.reshape(-1,1))

# now try LDA using two features
# make new model object
LDA = LinearDiscriminantAnalysis()
# fit the new model
LDA.fit(X_train[['petal_width', 'petal_length']], y_train)

# demonstrate predict_proba for two features
LDA.predict_proba(X_train[['petal_width', 'petal_length']])
```

Notes on the above code:

- This assumes you've already imported the iris data and done a train/test split
- For the single-feature example here, we used LDA with the "petal length" feature since it seemed to nicely separate the data among our three classes

28. Quadratic Discriminant Analysis (QDA)

`Lecture Notebooks/Supervised Learning/Classification/8. Quadratic Discriminant Analysis.ipynb`

Suppose we are trying to predict a variable y that can take on any of C possible classes using m features collected in a variable X

For LDA we assumed that $f_c(X)$ was the density function for a multivariate normal distribution with a class-specific mean vector μ_c and a covariance matrix common across all classes, Σ, so we had $X|y = c \sim N(\mu_c, \Sigma)$

If we're working with a sufficiently large sample, we can relax the assumption that the covariance matrix is the same across all C classes, so that we instead have $X|y = c \sim N(\mu_c, \Sigma_c)$

- Here Σ_c is the covariance matrix of $X|y = c$
- Conceptually, this means that we are no longer assuming the distributions of our various classes have equal variances
- When we relax this assumption the model is called Quadratic Discriminant Analysis (QDA)

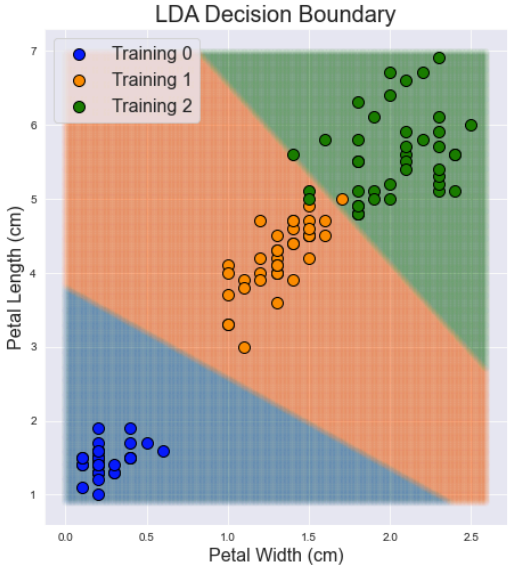
When we perform QDA we predict X^* to be the class which maximizes the following discriminant function:

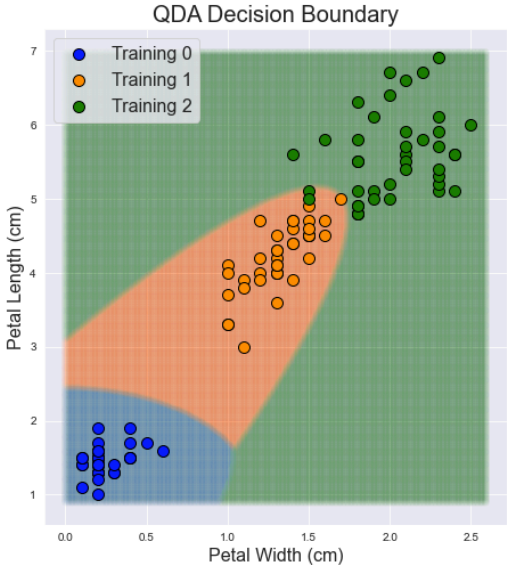
$$\delta_c(X^*) = -\frac{1}{2}(X^* - \mu_c)^T \Sigma_c^{-1}(X^* - \mu_c) - \frac{1}{2}\log(|\Sigma_c|) + \log(\pi_c)$$
$$= -\frac{1}{2}X^{*T}\sigma_c^{-1}X^* + X^{*T}\sigma_c^{-1}\mu_c - \frac{1}{2}\mu_c^T\sigma_c^{-1}\mu_c - \frac{1}{2}\log(|\Sigma_c|) + \log(\pi_c)$$

This is implemented in sklearn using QuadraticDiscriminantAnalysis

A key difference between LDA and QDA classifiers is the shape of the decision boundaries

- LDA decision boundaries are restricted to linear separation
  - In 2-D, this means that the feature plane is split using lines
  - In 3-D, using planes, and in higher dimensions, using hyperplanes
- QDA decision boundaries can be nonlinear, giving more flexibility
- See example figures below showing decision boundaries for the iris data set
  - The code to produce these plots can be found in the associated notebook





Recalling our prior discussion of the bias-variance tradeoff, LDA has more bias, while QDA has more variance

- Which you want to use in practice will depend on your data set

    - Particularly since you need fairly large amounts of training data to reliably use QDA
- In the iris data example above, the LDA model would probably generalize better
    - QDA seems to be overfitting, which makes sense given the relatively small data set

When to use LDA more generally:

- Because we don't have to estimate as many parameters, LDA works better for smaller data sets
    - In LDA we only have to estimate $m(M+1)/2$ covariances, while in QDA you have to estimate $Cm(m+1)/2$ of them
- LDA works better if you think your data can be mostly separated by linear decision boundaries
    - You might have an intrinsic reason to think this, or notice it when examining training data

When to use QDA more generally

- As mentioned above, QDA requires a large data set to properly fit
- QDA is preferable to LDA if you suspect that the data is better separated by a nonlinear decision boundary

Sample Code:

```
# import QDA package
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

# make the model object
QDA = QuadraticDiscriminantAnalysis()
# fit model object
QDA.fit(X_train[['petal_width', 'petal_length']], y_train)

# demonstrate predict_proba
QDA.predict_proba(X_train[['petal_width', 'petal_length']])
```

Notes on the above code:

- As with the LDA code, this assumes you've already imported the iris data and done a train/test split

29. Naïve Bayes Classifier

`Lecture Notebooks/Supervised Learning/Classification/9. Naive Bayes Classifier.ipynb`

For one last time, suppose we are trying to predict a variable y that can take on any of C possible classes using m features collected in a variable X using Bayes' rule to estimate $P(y = c|X)$:

$$P(y = 1|X = X^*) = \frac{\pi_c f_c(X^*)}{\sum_{i=1}^{c} \pi_l f_l(X^*)}$$

In order to make this estimate, we have to

1. Estimate the C values of π_c, which is relatively straightforward
2. Estimate C m-dimensional density functions $f_c(X)$, which is considerably less so

In LDA and QDA we made some strong assumptions regarding the form of the density functions which allowed us to take a hard estimation problem and make it much easier

- However, these are strong assumptions that could be way off

Rather than assuming a set form for the density, the naïve Bayes classifier takes a different approach, assuming that within a given class, c, each of the m features are independent

Thus, with $f_{c_j}(X_j)$ denoting the probability density function for X_j among observations of the c^{th} class, we write

$$f_c(X) = f_{c_1}(X_1) \times f_{c_2}(X_2) \times \cdots f_{c_m}(X_m)$$

And under this assumption, we have

$$P(y = 1|X = X^*) = \frac{\pi_c f_{c_1}(X_1^*) \times f_{c_2}(X_2^*) \times \cdots f_{c_m}(X_m^*)}{\sum_{i=1}^{c} \pi_l f_{l_1}(X_1^*) \times f_{l_2}(X_2^*) \times \cdots f_{l_m}(X_m^*)}$$

Assuming independence between the feature variables allows us to turn the difficult problem of estimating an m-dimensional probability distribution (which involves estimating both m marginal distributions and a joint distribution) into a problem where we have to estimate just m independent probability distributions

- Much more tractable!

When it comes to estimating the f_{c_j} we typically assume some kind of distribution and then estimate the parameters for that distribution accordingly, for example:

- If X_j is quantitative, we typically assume it is a normal distribution
    - Note that this differs from LDA and QDA because those do not assume independence, hence their associated covariance matrices Σ or Σ_c
- If X_j is categorical, we could just use a Bernoulli distribution (think biased coin toss) estimating the value of p using the proportion of observations where $y = c$ for each possible value of X_j

Assuming independence is not always a good assumption in the sense that some of the features may well be related to one another

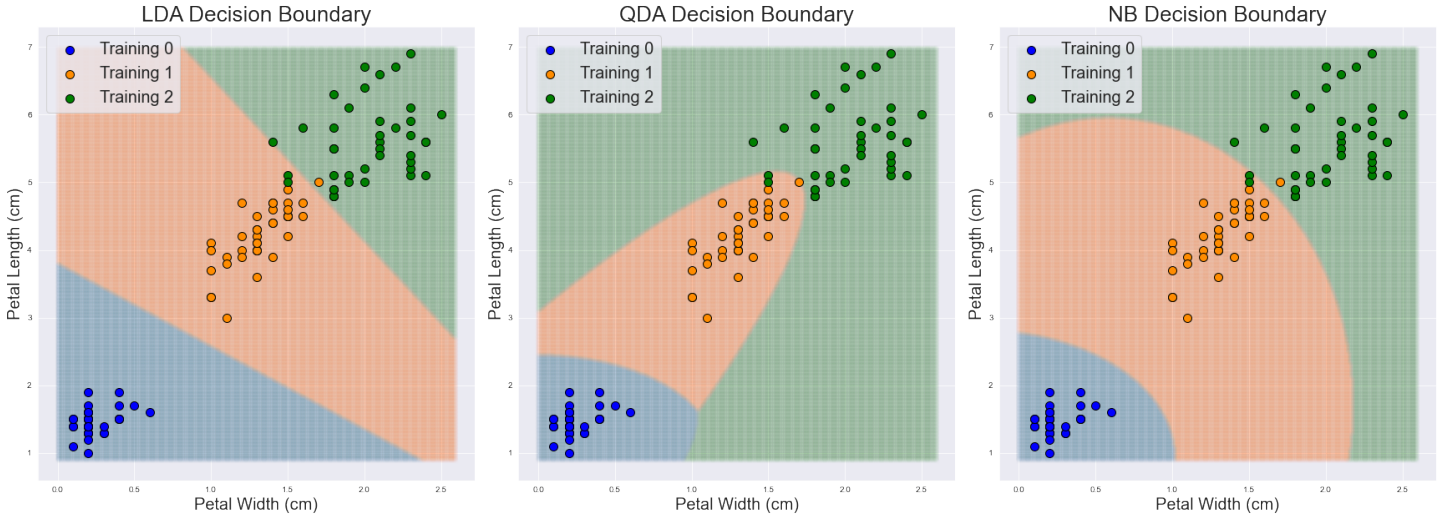
- However, even if the assumption does not hold, we can still get decent-to-good classifiers using a naïve Bayes classifier
- This can be particularly true if we have insufficient data to reasonably estimate a joint probability distribution
- We can think of this assumption as adding bias to our model

Naïve Bayes is implemented in sklearn with a few different methods found in the naive_bayes module

- When working with quantitative features like the petal length and width features found in the iris data set, we'll use the GaussianNB model
- If we instead had features of categorical variables, we'd likely use the BernoulliNB model

As with QDA, the naïve Bayes classifier allows for non-linear decision boundaries

- Additionally, by introducing some bias we are likely producing a model that generalizes better than that of QDA given the relatively small data set in this example
- See associated notebook for the code to produce these plots for the iris data set



Sample Code:

```
# import GaussianNB
from sklearn.naive_bayes import GaussianNB

# make the model
nb = GaussianNB()
# fit the model
nb.fit(X_train[['petal_width', 'petal_length']], y_train)

# demonstrate predict_proba
nb.predict_proba(X_train[['petal_width', 'petal_length']])
```

Notes on the above code:

- As in prior lessons, this assumes you've already imported the iris data and done a train/test split

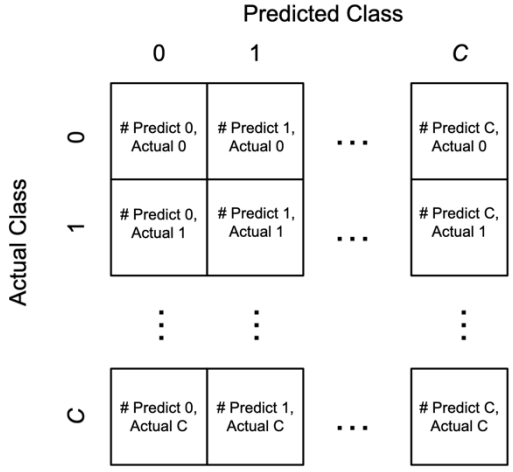
30. Multiclass Classification Metrics

`Lecture Notebooks/Supervised Learning/Classification/10. Multiclass Classification Metrics.ipynb`

One approach (that is admittedly bit of a cop out) would be to simplify things back down to a binary classification and then use the same metrics discussed previously

- For example, if you have NFL play data coded as a successful/unsuccessful run or pass play, you might be able to simplify things so that you only have to classify successful vs. unsuccessful plays
- Obviously this is application-dependent and will not work for the general case

More generally, the confusion matrix we discussed earlier naturally extends to additional classes like so



Although we do lose our easily interpreted nomenclature from earlier when we move to the multiclass case

We can obtain a confusion matrix for a multiclass classifier using sklearn's confusion_matrix function, the same as for a binary classifier

Another potentially useful metric for multiclass classification is cross-entropy, sometimes referred to as log-loss

- This is essentially a measure of how well two probability distributions align with one another

Here we compare the "distribution" of the sample to the estimated probability distribution from the model

- For the sample, we take "distribution" to mean a set of indicator functions $y_c = 1_{y=c}$

So for each observation, i, we compute

$$-\sum_{c=1}^{C} y_{c,i} \log(p_{c,i})$$

where C is the total number of possible classes and $p_{c,i}$ is the model probability that observation i is of class c

- Essentially the $p_{c,i}$ terms are found using the predict_proba output for your given model
- The total cross-entropy for the sample is the total sum for all i_{tot} observations

The only term that contributes to the sum in the above expression is the one corresponding to the class that observation i actually is, for example let's say l

- If we know that the class of i is l, then observation i contributes $-\log(p_{l,i})$ to the total cross-entropy
- When $p_{l,i}$ is closer to 1, $-\log(p_{l,i})$ is closer to 0
    - So when our model is confident in its correct prediction, it contributes little to the cross-entropy
- Conversely, as $p_{l,i} \to 0$, then $-\log(p_{l,i}) \to \infty$

- When our model is confidently incorrect, it contributes significantly to the cross-entropy
- Thus, cross-entropy is notable because it punishes models that are confidently incorrect

Since we want to assign observation i to its actual class, what we want is a value of $p_{i,j}$ that is close to 1

- The closer to 1 each of the $p_{i,i}$ terms are, the closer to 0 our total entropy is
- Thus, a good model is one with low cross-entropy

Cross-entropy can be coded up manually, but in practice you'll likely want to use sklearn's log_loss function

- Note that in practice you'll likely want to use cross validation when checking cross-entropy
- See associated notebook for a worked out example doing this CV exercise for the LDA, QDA, and naïve Bayes classifiers trained on the iris data set we've been looking at in prior lessons

Sample Code:

```
# import confusion matrix
from sklearn.metrics import confusion_matrix

#get a confusion matrix (no different from binary classification)
confusion_matrix( y_train,  LDA.predict(X_train[['petal_width', 'petal_length']]) )

# make a confusion matrix dataframe
pd.DataFrame( confusion_matrix(y_train, LDA.predict(X_train[['petal_width', 'petal_length']])),
columns = ['Predicted 0', 'Predicted 1', 'Predicted 2'], index = ['Actual 0', 'Actual 1', 'Actual 2'] )

# let's do a manual cross-entropy calculation
# generate the ycs
ycs = pd.get_dummies(y_train).to_numpy()
# generate the pcs
pcs = LDA.predict_proba(X_train[['petal_width', 'petal_length']])

# take the sum of each rows product
- np.sum(ycs * np.log(pcs), axis=1)
# now sum that sum to get the total cross-entropy for this LDA model
np.sum(- np.sum(ycs * np.log(pcs), axis=1))

# now do this with sklearn, starting with an import
from sklearn.metrics import log_loss
# implement log_loss on the training set
log_loss(y_train, pcs, labels = [0,1,2], normalize=False)
```

Notes on the above code:

- This assumes you've already fit an LDA classifier, and it'd work just as well with a different classifier
- We gave log_loss the pcs output since we'd already calculated it in the sample code, but if we hadn't done so we could have simply put the predict_proba call there
- The labels parameter is just a list of the labels in your data set
- If we don't set normalize to False we instead get the total cross-entropy divided by the number of elements in y_train, an average of sorts
  - In practice it won't make a difference which convention we use as long as we're consistent

31. Principal Components Analysis I

`Lecture Notebooks/Unsupervised Learning/Dimensionality Reduction/1. Principal Components Analysis I.ipynb`

Why even bother with dimensionality reduction?

- Perhaps the dimension is too large for your algorithm to run efficiently
- Sometimes you want to get rid of (or mitigate) some noise-inducing variables
- Other times you just want to better visualize the data in 2D or 3D

In the next few lessons we will be discussing perhaps the most popular dimensionality reduction technique in data science, principal component analysis (PCA)

When you reduce the dimension of a data set you are inherently losing information, so you want to ensure that you do it in a way that "retains as much information as possible"

- PCA tackles this problem in a very statistical manner

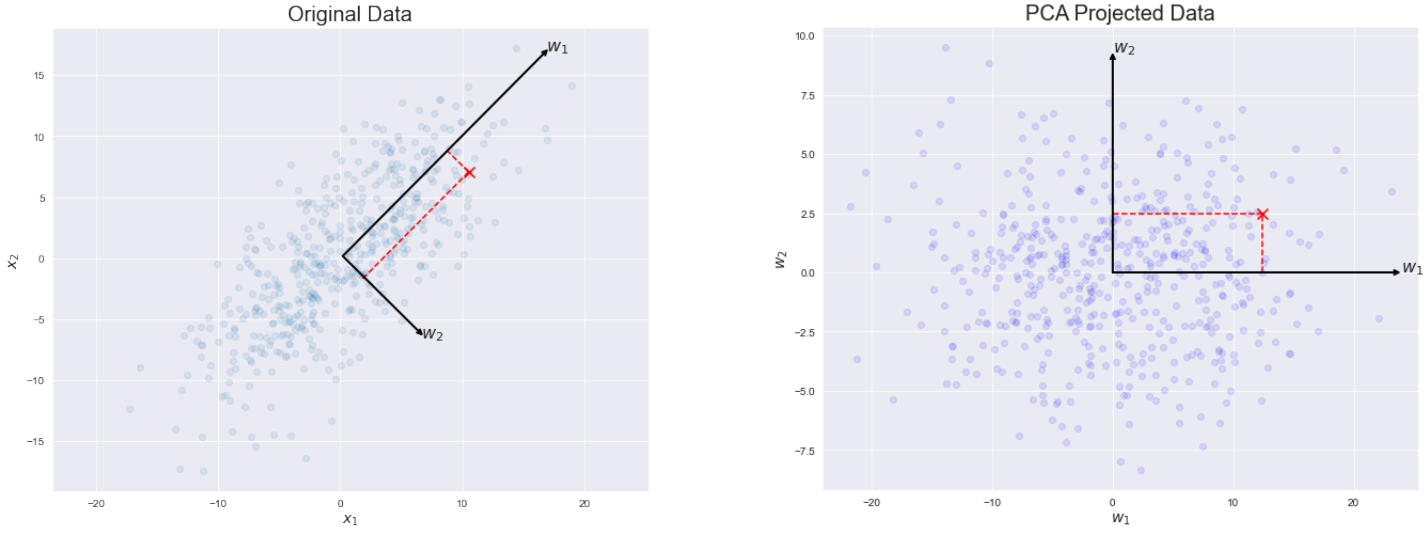
There's an idea in statistics that the information of a data set is located within that data set's variation, and thus when you reduce the dimension of a data set, you want to project your data onto a lower dimensional space that captures as much of the original variance in the data as possible

- Thinking in terms of optimization, your goal is to project into a lower dimensional hyperplane in a way that maximizes variance

Here's a heuristic algorithm for PCA

1. Center your data so that each feature has 0 mean, this is done for convenience
2. Find the direction in space along which projections have the highest variance
    - This is the first principal component
3. Find the direction orthogonal to the first principal component that maximizes variance
    - This is the second principal component
4. Continue in this way, so that the k^{th} principal component is the variance-maximizing direction orthogonal to the previous $k-1$ components

We can code this up using sklearn's PCA function, allowing us to illustrate the method in the example below



The two vectors w_1 and w_2 drawn above are called the component vectors of the PCA

- To transform our data, we take the scalar projection of each observation onto the component vectors, as illustrated by the sample point shown by the red x marker

Now let's briefly summarize the mathematical details underpinning PCA

Suppose we have n observations of m features, let $X_1, X_2, \dots, X_m$ be n by 1 vectors containing the observations of each of the m features, and for ease of notation assume each has been centered to have mean 0

- We'll restrict ourselves to the case of finding the first principal component, noting that the others can be found in a similar fashion

Let $X = (X_1|X_2| \dots |X_m)$ be an n by m feature matrix, and note that our goal is to find $w = (w_1, w_2, \dots, w_m)^T$ with $\|w\| = 1$ such that $\text{Var}(w_1X_1 + w_2X_2 + \cdots + w_mX_m) = \text{Var}(Xw)$ is maximized

- Note that because $\|w\| = 1$, Xw is a vector of scalar projections of the rows of X onto w

Because we have centered the columns of X, we have $\text{Var}(Xw) = E(w^TX^TXw) = w^TE(X^TX)w = w^T\Sigma w$, where Σ is the covariance matrix of X, and our constrained optimization problem is now:

$$\text{optimize } f(w) = w^T\Sigma w, \text{constrainted to } g(w) = w^Tw - 1 = 0$$

Using the method of Lagrange multipliers and some matrix calculus, we find

$$\partial_w\big(w^T\Sigma w - \lambda(w^Tw - 1)\big) = 2\Sigma w - 2\lambda w$$

Setting this to 0 we obtain $\Sigma w = \lambda w$, the standard eigenvalue setup

Thus, the vector w that maximizes variance is an eigenvector corresponding to the largest eigenvalue of the covariance matrix X

- This vector is known as the first principal component

Note that because Σ is an $m \times m$ real positive symmetric matrix, it has a set of m eigenvalues (assuming $n > m$) with orthogonal eigenvectors

- Thus the remaining principal component vectors are the eigenvectors corresponding to the eigenvalues of Σ in decreasing order

For each weight vector, w, we call $\text{Var}(Xw)$ the explained variance due to the principal component w

- We can think of this as the amount of variance in X explained by the principal component w
- This can be accessed in sklearn with explained_variance_

It can be useful to think of this in terms of the portion of $\text{Var}(Xw)$ explained by the principal direction, w

- As we will see shortly, this can be helpful in determining how many components we should project down to
- This value can be accessed in sklearn with explained_variance_ratio

Although we didn't do so in this example, we typically need to scale our data prior to fitting a PCA model

- This is because the variance of a large scale feature is inherently larger than the variance of a small scale feature
- So if we have data with vastly differing scales, we will not be recovering the "hidden structure" of the data, but rather showing what columns have the largest scale
- A common scaling approach in practice is to run the data through StandardScaler first

Sample Code:

```
# PCA is stored in decomposition
from sklearn.decomposition import PCA
```

```
# make the PCA object, projecting down to 2-D
pca = PCA(2)
# fit the data
pca.fit(X)

# transform gets you the PCA transformed values for your data (in this case x1/x2 --> w1/w2)
fit = pca.transform(X)

# demonstrate .components utility
pca.components_

# use it to get PCA component vectors w1 and w2 (these are vectors, so each will be itself a list)
w1 = pca.components_[0]
w2 = pca.components_[1]

# demonstrate explained_variance_
pca.explained_variance_
# demonstrate explained_variance_ratio_
pca.explained_variance_ratio_

# get the explained variance for the two PCA components (these are just scalar numbers)
expvar_w1 = pca.explained_variance_[0]
expvar_w2 = pca.explained_variance_[1]
```

Notes on the above code:

- Here we have gone from 2D data to a 2D PCA, but in typical applications you'll more likely be using PCA to reduce the dimensions
    - For example, looking only at w_1 in this illustrative exercises
- Note that the components_, explained_variance_, and explained_variance_ratio_ attributes return arrays of values associated with however many components you've built your PCA object with (in this case 2)
    - These arrays start with the first principal component in the 0^{th} index

32. Principal Components Analysis II

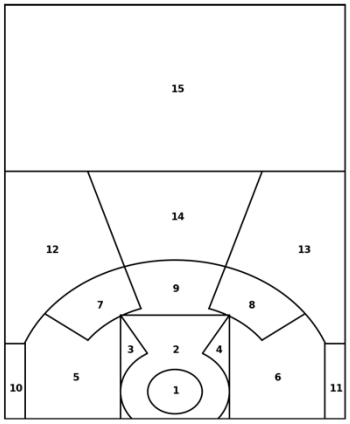
`Lecture Notebooks/Unsupervised Learning/Dimensionality Reduction/2. Principal Components Analysis II.ipynb`

While PCA can be quite useful, you tend to lose some of the interpretability of the data set's original features

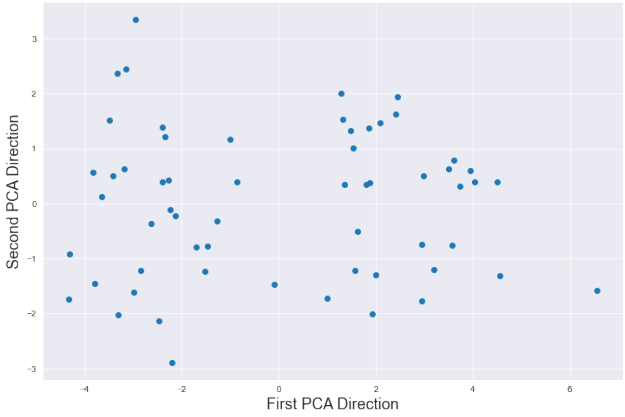
- In most applications, principle component 1 is usually more difficult to understand than something like dollars, units sold, points scored, etc.
- Luckily, we can use the component vectors, the *w*s, to help us understand what each principal component direction is capturing

To help explain how to do this in practice, we'll use a new data set stored in nba_team_shots.csv that tracks the shot distribution for NBA teams in the 2000-01 and the 2018-19 seasons

- Note that this data set tags shot location using the 15 unique court zones illustrated and labeled below
- In this example we'll project this 15-dimensional data set down into 2D using PCA



ID	Zone Name
1	Restricted Area
2	In the Paint, Center
3	In the Paint, Left
4	In the Paint, Right
5	Mid-Range, Left
6	Mid-Range, Right
7	Mid-Range, Center-Left
8	Mid-Range, Center-Right
9	Mid-Range, Center
10	Left Corner 3
11	Right Corner 3
12	Above the Break 3, Left
13	Above the Break 3, Right
14	Above the Break 3, Center
15	Backcourt

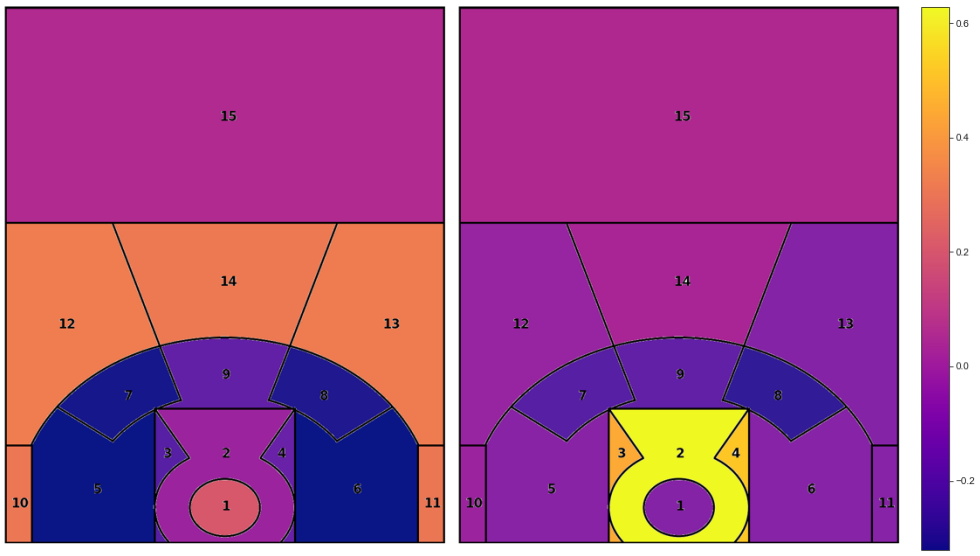


We can apply PCA to the data set easily enough to obtain the plot on the right, but what does it mean?

- Let's use the component vectors to find out!
- See associated notebook for the heat map code used to obtain the colors below

	component_1	component_2
zone_6_perc_att	-0.320828	-0.059241
zone_5_perc_att	-0.320349	-0.051844
zone_7_perc_att	-0.308441	-0.175883
zone_8_perc_att	-0.294858	-0.256153
zone_3_perc_att	-0.172937	0.444518
zone_9_perc_att	-0.148686	-0.152251
zone_4_perc_att	-0.130365	0.516082
zone_2_perc_att	-0.003040	0.627804
zone_15_perc_att	0.059863	0.054102
zone_1_perc_att	0.208658	-0.064286
zone_11_perc_att	0.301864	-0.057806
zone_10_perc_att	0.308146	-0.004666
zone_14_perc_att	0.314199	0.038328
zone_13_perc_att	0.323201	-0.062398
zone_12_perc_att	0.327066	-0.014621

PCA Component 1 PCA Component 2



We see that the first PCA component has significant positive weights for zones 12, 13, 14, 10, and 11, while it has significant negative weights for zones 6, 5, 7, and 8

- So teams with values > 0 in this PCA direction are making more 3-point shots than mid-range shots

For the second PCA component, on the other hand, we find significant positive weights for zones 2, 4, and, 3, and significant negative weights for zones 8, 7, and 9

- So teams with values > 0 in this second PCA direction are making more shots in the paint than center-court mid-range shots

Sample Code:

```
# import the data
shots = pd.read_csv("../../../Data/nba_team_shots.csv")

# import PCA and scaler functions
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# make associated objects
scaler = StandardScaler()
pca = PCA(n_components=2)

# fit the pca and get the transformed data
X_scaled = scaler.fit_transform(shots[shots.columns[3:]])
pca.fit(X_scaled)
fit = pca.transform(X_scaled)

# make a data frame for the component vectors so we can analyze them
component_vectors = pd.DataFrame( pca.components_.transpose(),
                    columns = ['component_1', 'component_2'], index = shots.columns[3:] )

# look at features and how they are weighted, sorted by their impact on the first PCA direction
component_vectors.sort_values('component_1')
```

Notes on the above code:

- We use shots.columns[3:] because we only care about the shot percentages in each zone, and the first three entries are instead related to team and season identification
- We need to include transpose() when making our illustrative dataframe because pca.components_ by default gives us our vectors as rows and we need them as columns here

33. Principal Components Analysis III

`Lecture Notebooks/Unsupervised Learning/Dimensionality Reduction/3. Principal Components Analysis III.ipynb`

A key question with PCA is how many components you should use, and as is so often the case in data science, the answer is dependent upon your use case

- If you are interested in using PCA to produce a visualization of data, you likely want 2 or 3 components
- On the other hand, if you're using PCA to battle collinearity for a regression you'll want to use as many components as you have columns

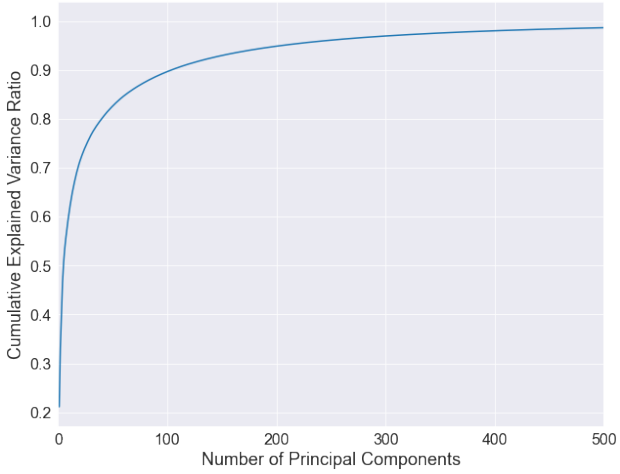
When the answer is not obvious from your use case, you can turn to the explained variance ratio

We'll explore this concept using the "labeled faces in the wild" data set

- Each observation in this data set represents a publicly sourced 87 by 65 grayscale image of a relatively well-known public figure
    - This translates to $87 \times 65 = 5{,}655$ features for each observation
- This number of features could be computationally expensive for some of the supervised learning algorithms we have learned up to this point, for example *k*-nearest neighbors
    - So let's look at using PCA to reduce the number of dimensions

We previously described the explained variance ratio as giving the portion of the original variance of X explained by each principal component direction

- We often look at the explained variance curve, which plots the cumulative explained variance ratio against the number of directions you have considered



What we typically do is look for the elbow in the explained variance curve

- The elbow is where the amount of added variance ratio starts to rapidly decrease
    - In this example (shown above) it appears that the elbow occurs around 100 components
- We take this as an indication that adding additional components will start to have diminishing returns

Alternatively, we can simply set a cumulative explained variance ratio that we want to achieve and let sklearn find the number of PCA components necessary to attain it

- We do this by simply setting n_components to be our desired cumulative explained variance ratio

In general, we can think of PCA as a way to compress the data contained in X, and we can use a little linear algebra to help us see how well the compressed data compares to the original

Suppose that we are in $\mathbb{R}^2$ and that vector v and vector u are perpendicular, then for any vector x we can write

$$x = \text{proj}_u(x) + \text{proj}_v(x)$$

Suppose that we have an observation X^*, then for any principal component vector w_l we have that

$$\text{proj}_{w_l}(X^*) = (X^* \cdot w_l)w_l \equiv \tilde{X}_l^* w_l$$

If we define $\tilde{X}_l^*$ to be the l^{th} principal value for observation *, and if we take L to be the total number of principal components, then we can approximate the original X^* as

$$X^* \approx \tilde{X}_1^* w_1 + \tilde{X}_2^* w_2 + \cdots + \tilde{X}_L^* w_L$$

When we apply this to our faces data set, examining how well images are reconstructed using PCA models of varying size, we see how an extra bit of explained variance can make a big difference

Sample Code:

```
# get and load the labeled faces data
from sklearn.datasets import fetch_lfw_people
people = fetch_lfw_people(min_faces_per_person=20, resize=0.7)
image_shape = people.images[0].shape
# import PCA
from sklearn.decomposition import PCA

# need to scale the facial imagery data
X = people['data']
```

```
# this scales it so that that the max value in a pixel is 1
X = X/255

# make and fit the PCA model
pca = PCA()
pca.fit(X)

# get the explained variance ratio
pca.explained_variance_ratio_

# make another PCA model, this time focused on attaining a 95% cumulative explained variance ratio
pca = PCA(n_components=.95)
pca.fit(X)

# fit a PCA with 3000 components to demonstrate image reconstruction
pca = PCA(n_components=3000)
pca.fit(X)

# get the projection onto the lower dimensional PCA space
X_tilde = pca.transform(X)

# get the reconstruction for the 0th index image
X_tilde[0,:3000].dot(pca.components_).reshape(image_shape)
```

Notes on the above code:

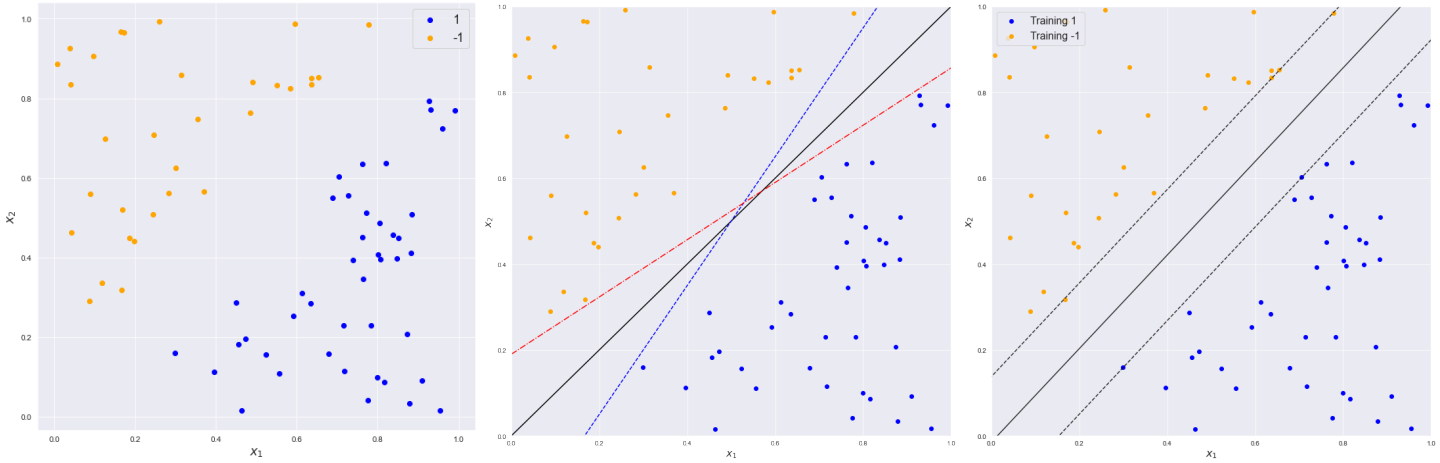
- By leaving the PCA() call blank when we make the model (instead of specifying the number of components), it will fit the maximum number of possible components
- If instead we give PCA() a fraction, it will fit the number of components needed to attain a cumulative explained variance ratio equal to that fraction
- It is standard practice when working with image data to scale pixel values such that they can vary from 0 to 1, as we have done here

34. Linear Support Vector Machines

`Lecture Notebooks/Supervised Learning/Classification/11. Linear Support Vector Machines.ipynb`

Linear support vector machines are a particular branch of support vector machines that are used to separate data that you suspect have linear, or nearly linear, decision boundaries

We will start with what are sometimes referred to as maximal margin classifiers, looking first at an example



Here you can clearly separate the two classes with a simple straight line, and more generally with what's known as a hyperplane

- Note that for any high dimensional space, a subspace that is one dimension lower is a hyperplane
- So in 1-D ($\mathbb{R}^1$) a hyperplane is simply a point, in 2-D ($\mathbb{R}^2$) a hyperplane is a line, in 3-D ($\mathbb{R}^3$) a hyperplane is a 2-D plane, and in $\mathbb{R}^n$, it is an $n-1$ subspace

We can draw here a number of lines that cleanly separate the data sets (see middle plot above), but we want to identify the boundary that will best generalize to future data sets

- Since the red and blue lines are very near the training data at points, it is likely that new observations would deviate to the wrong side of the decision boundary due to random noise

One approach is to draw a hyperplane that maximizes the total distance from all points to the hyperplane

- Essentially we want to draw a hyperplane, find the minimum distance from the points to the hyperplane (known as the margin), then make that as large as possible (maximize it)
- Hence the name maximal margin classifier

Given a binary variable, $y \in \{-1,1\}$, and a set of m features stored in the columns of a feature matrix X, we can find the maximal margin classifier, if it exists, by solving the constrained optimization problem

$$\text{find } \beta \text{ and maximal } M, \text{subject to } \|\beta\|_2^2 = 1 \text{ and } y^{(i)}\left(X^{(i)}\beta\right) \geq M \; \forall i = 1, \dots, n$$

where $\beta = (\beta_0, \beta_1, \dots, \beta_m)^T$ is a coefficient vector and X has been extended to include a column of ones

- It is possible to solve this optimization problem, but we won't address the details of doing so here
  - If interested, this solution can be found in the Elements of Statistical Learning book

Now, $y^{(i)}\left(X^{(i)}\beta\right) \geq M$ may look weird, but $X\beta = 0$ is a formula that defines a hyperplane, for example:

$\beta_0 + \beta_1 X_1 + \beta_2 X_2 = 0$ is the formula for a line in 2-D, and

$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 = 0$ is the formula for a plane in 3-D

So $y^{(i)}\left(X^{(i)}\beta\right) \geq M$ just establishes that we want all of our points to fall outside a margin M on either side of the hyperplane $X_i\beta = 0$

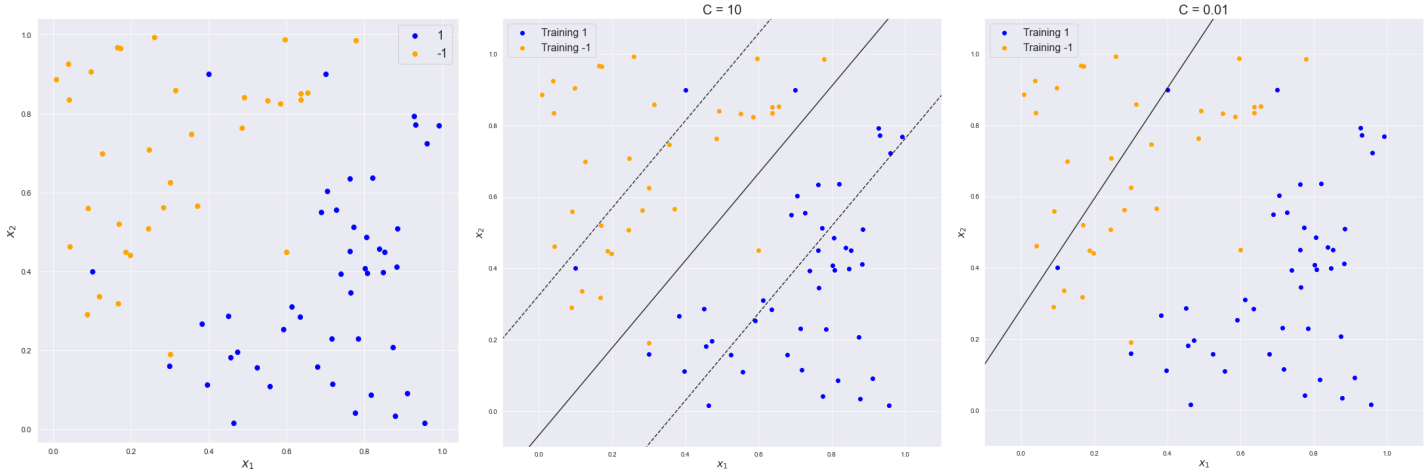
The above right example plot shows a practical solution, with the decision boundary "hyperplane" shown by the solid black line, and the dotted lines making a distance M away from it

- You might notice that some of the points touch the margin lines
- Such observations are known as the support vectors because they "support" the separating line, in the sense that if we move these points slightly, the line will move as well

Okay so we now have an algorithm that can separate groups that are separable by hyperplanes, but there are two looming possible issues:

- There's no guarantee our data will be cleanly separable by a hyperplane
- The maximum margin classifier may be too sensitive to training data

For example, what if our data set from above had a few outliers in each class?



While this new version of the data isn't linearly separable, it is *almost* linearly separable, and if we're willing to accept this imperfect separation we can still use the maximal margin classifier as our guide for a new algorithm called the soft margin or support vector classifier

The maximal margin classifier is a hard margin classifier, meaning that instances of class 0 are not allowed to cross the decision boundary over into the area occupied by class 1, but we can relax this rule, modifying the associated constrained optimization problem so that we aim to

$$\text{find } \beta \text{ and maximal } M, \text{subject to } \|\beta\|_2^2 = 1 \text{ and } y^{(i)}\left(X^{(i)}\beta\right) \geq M \; \forall i = 1, \ldots, n$$

$$\epsilon^{(i)} \geq 0, \sum_{i=1}^{n} \epsilon^{(i)} \leq \frac{1}{C}$$

As before, we'll leave the details of solving this problem in the Elements of Statistical Learning book

Here $\epsilon^{(i)}$ are referred to as slack variables because they control how much observation i can violate the margin

- If $\epsilon^{(i)} = 0$, then observation i is on the correct side of the margin
- If $0 < \epsilon^{(i)} < 1$, then observation i is on the wrong side of the margin, but correct side of the hyperplane
- If $\epsilon^{(i)} > 1$, then observation i is on the incorrect side of the hyperplane

Meanwhile, C is a hyperparameter that you tune, typically using cross-validation

- The value of C we choose can be thought of as our "budget" for the slack variables $\epsilon^{(i)}$
- Larger values of C lead to a smaller budget, so we approach the maximum margin classifier discussed earlier, while smaller values of C produce softer margins

- As demonstrated above, changing the value of C can have a significant impact on the location of your decision boundary
    - Note the differences between the $C = 10$ and $C = 0.01$ boundaries

For the support vector classifier, we consider any point that touches or is on the wrong side of the margin (that is, those for which $\epsilon^{(i)} > 0$) a support vector

- As before, these are called support vectors because they "support" the decision boundary in a sense
- Slight changes to these points will shift the decision boundary, while slight changes to non-support vector points will not
    - However, large changes to previously non-support vector points may move them into the regime where they become support vectors themselves

We code up both the maximal margin classifier and the support vector classifier using sklearn's LinearSVC

We have presented support vector machines as a binary classification algorithm here, but in practice they can also be used for multiclass classification

- This is true for both the linear support vector machines covered in this lesson and the general support vector machines we'll cover next lesson
- sklearn approaches multiclass support vector machines with a one vs one approach in which you train a unique support vector machine classifier for each possible pair of classes
- If you had three classes 1,2,3, for example, you would train a 1 or 2 classifier, a 1 or 3 classifier, and a 2 or 3 classifier
- For a problem with C classes you will in general train $C(C-1)/2$ different support vector machines

Sample Code:

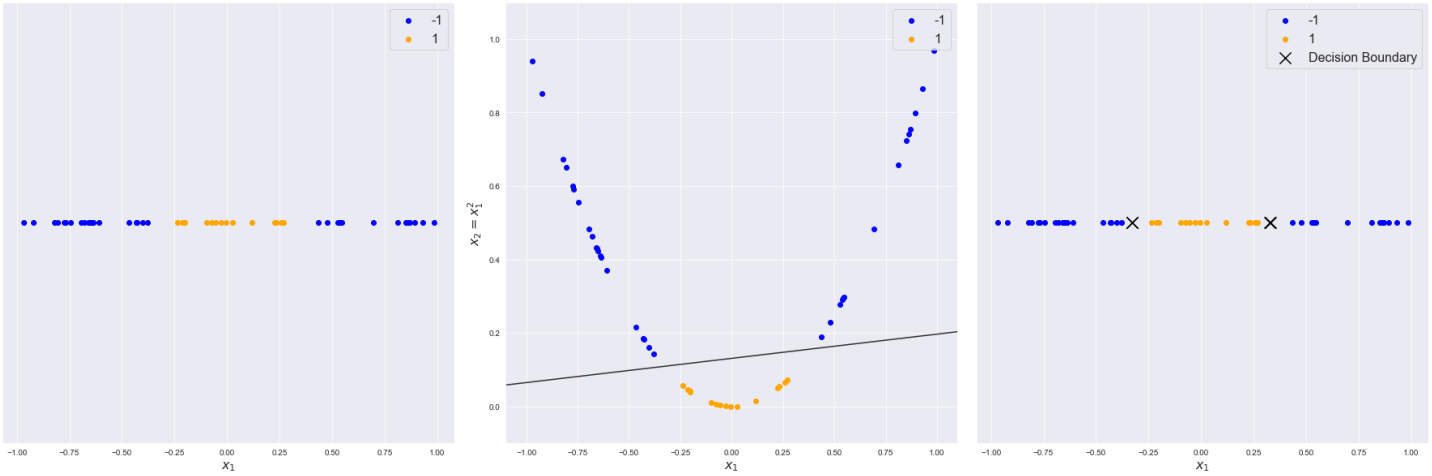
```
# import LinearSVC
from sklearn.svm import LinearSVC

# make model, first for the maximal margin classifier with its large C
max_margin = LinearSVC(C = 1000, max_iter = 100000)
# fit model
max_margin.fit(X, y)

# make model, this time demonstrating a more relaxed support vector classifier with a moderate C
svc = LinearSVC(C = 10, max_iter = 100000)
# fit model
svc.fit(X, y)
```

Notes on the above code:

- Note that we code up the maximal margin classifier as a special case of the support vector classifier's LinearSVC tool by using a large C value
- Since LinearSVC does not have a predict_proba method, if you needed prediction probabilities (for example if you wanted to look at an ROC curve), you'd need to use the SVC package (introduced in the next lesson) with the arguments kernel='linear' and probability=True

Note that, while we have examined SVMs in the context of classification, they can also be used for regression

We previously discussed linear support vector machines which look to find a hyperplane that separates your dataspace into two halves, but what if you're working with data that can't be separated by a hyperplane?



The 1-D data set shown above, for example, cannot be separated by any single point (the hyperplane for a 1D space), but we *can* find a linear separation if we "lift" it into a 2-D space using the transformation $x_2 = x_1^2$

- We can then convert the intercepts from this 2D decision boundary line into 1D decision boundaries

This trick is the essence of the more general support vector machine – we take data in a lower dimension and somehow embed it into a higher dimension, one where it is hopefully close to linearly separable

- One clear potential issue is how to choose an appropriate embedding, as we could easily end up with an enormous magnitude of potential features, which would lead to impractical computational costs
- Fortunately, support vector machines have a nice way of handling this problem

While we have not discussed how the solution to the support vector classifier is computed, because it is a bit long and technical, it turns out that it solely involves the inner products between the training observations

- Note that the inner product between two vectors $a, b \in \mathbb{R}^m$ is computed as $\langle a, b \rangle = \sum_{i=1}^{m} a_i b_i$
    - For our purposes we can think of the inner product as a dot product, so $\langle a, b \rangle = a \cdot b$

If we have some function ϕ that takes our observations into a higher dimensional space in order to separate them, then we'll need to compute $\langle \phi(X^*), \phi(X^\#) \rangle$ for pairs of observations X^* and $X^\#$

- ϕ can ultimately take our lower dimensional data into very high (practically infinite) dimensional space, which would make it nearly (or actually) impossible to compute the inner products necessary to estimate the separating boundary
- But what if we didn't need to actually compute $\langle \phi(X^*), \phi(X^\#) \rangle$ in the higher dimensional space?

For example, let's say our data has two features x_1 and x_2 and our ϕ is such that

$$\phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1 x_2 \\ x_2^2 \end{pmatrix}$$

For two vectors a and b, the inner product between $\phi(a)$ and $\phi(b)$ is

$$\langle \phi(a), \phi(b) \rangle = a_1^2 b_1^2 + 2a_1 b_1 a_2 b_2 + a_2^2 b_2^2 = (a_1 b_1 + a_2 b_2)^2 = \langle a, b \rangle^2$$

So even though we knew what the map ϕ was to the higher space here, we didn't actually need to know $\phi(a)$ or $\phi(b)$ in order to compute the dot product $\langle \phi(a), \phi(b) \rangle$

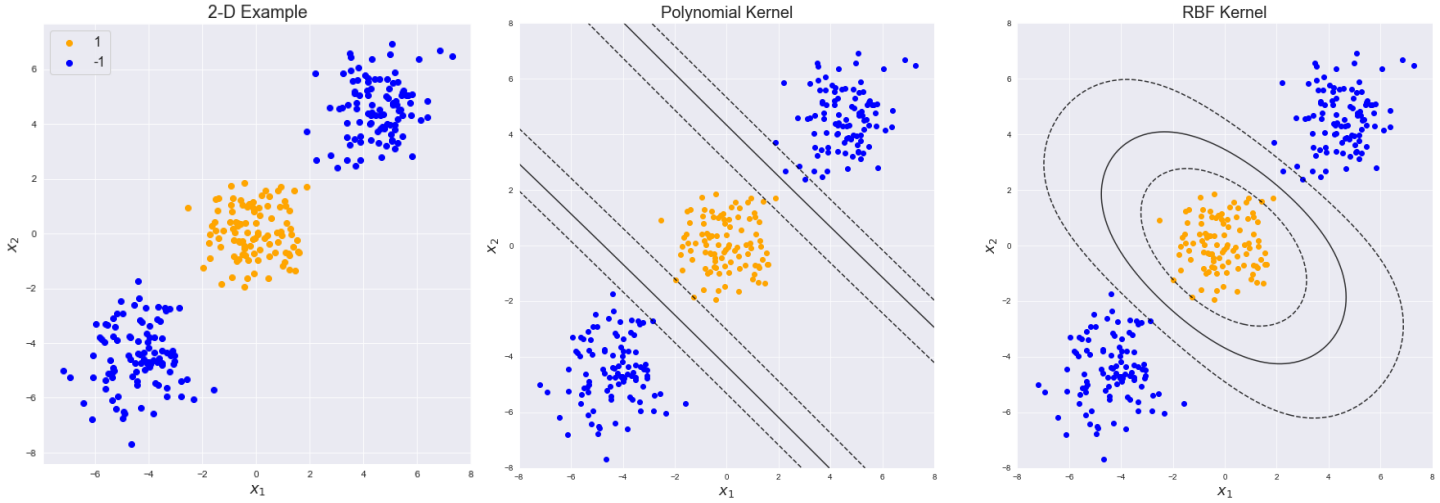
In this context we say a map ϕ has a kernel function $\boldsymbol{K}$ if $\langle \phi(a), \phi(b) \rangle = \boldsymbol{K}(a, b)$, where $\boldsymbol{K}$ is only a function of a and b in the original feature space

- So for the example above the kernel function is $\boldsymbol{K}(a, b) = (\langle a, b \rangle)^2$

Common kernel functions for support vector machines are:

- Linear: $\boldsymbol{K}(a, b) = \langle a, b \rangle^2$
  - This gives the linear classifiers we looked at earlier
- Polynomial: $\boldsymbol{K}(a, b) = (\gamma \langle a, b \rangle + r)^d$
  - Our example ϕ above is an example of this, and one was used for our earlier 1-D data example
- Gaussian Radial Kernel (Gaussian RBF): $\boldsymbol{K}(a, b) = \exp(-\gamma \|a - b\|^2)$, where $\|\circ\|$ is the Euclidean norm
- Sigmoid: $\boldsymbol{K}(a, b) = \tanh(\gamma \langle a, b \rangle + r)$
- In these functions γ, r, and d are hyperparameter to be tuned using something like cross-validation
  - These are in addition to the margin hyperparameter C, which is still pertinent here

We implement support vector machines in sklearn using SVC, and when doing so we can specify different kernels and compare their resulting decision boundaries



Sample Code:

```
# import SVC
from sklearn.svm import SVC

# make two svc models to compare between kernels
svc_poly = SVC(C=10, kernel='poly', degree=2)
svc_rbf = SVC(C=10, kernel='rbf')
# fit the models
svc_poly.fit(X, y)
svc_rbf.fit(X,y)
```

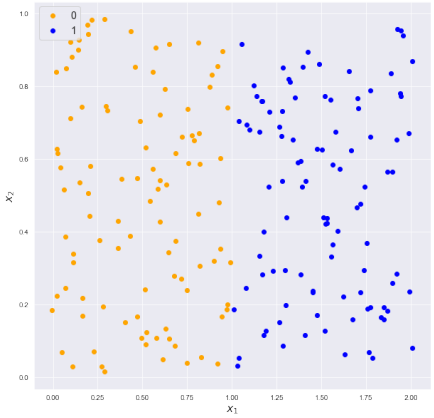
Notes on the above code:

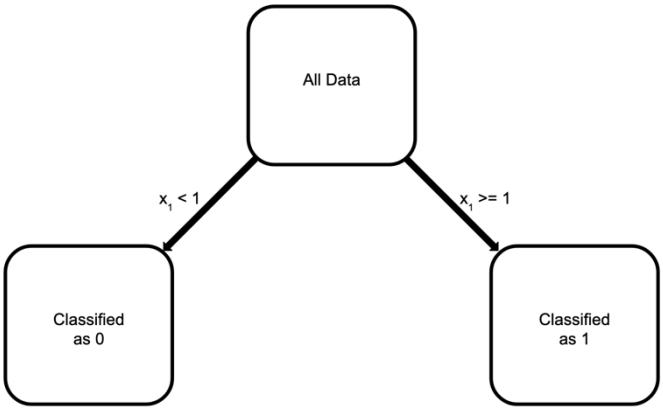
- An svc model with the kernel='poly' option can also be used to recover the same classifier we looked at in the 1-D example earlier in the lesson, although you need to use reshape(-1,1) again in that context
- If you have an application where linear SVC is likely to be appropriate, though, you should use the LinearSVC package since it will be more efficient than the more general SVC package introduced here

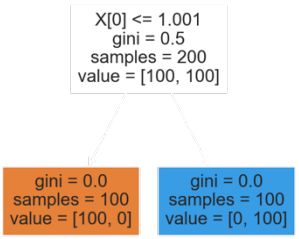
36. Decision Trees

`Lecture Notebooks/Supervised Learning/Classification/13. Decision Trees.ipynb`

If you have the data set plotted below, one choice for a classifier would be to say that for points left of the $x_1 = 1$ line, the class is $y = 0$, while for points right of the $x = 1$ line, it is $y = 1$







While this is an extremely simple example, it shows the basic idea behind decision trees

- You look at a feature, make a cut point that minimizes some measure of wrongness, and keep going until you reach some stopping criterion

In sklearn we fit decision trees using the model object DecisionTreeClassifier, and we can also use tools from the tree package to visualize the resulting fit algorithm

- The plot shown on the right is the logic tree built by the decision tree algorithm
- To classify a new observation we start at the *root node* up top
- If the observation satisfies the logic statement at the top we go left and are classified as a 0, otherwise we go right and are classified as 1
- The two *children* of the root node are known as *leaf nodes* or *terminal nodes* because they have no children of their own
    - At this stage we take the majority class contained in that node to be the classifier's prediction for future samples
- Ultimately, this recovers the same rule we had identified by eye, as it should for this simple example

Each node in the decision tree plot includes a number of different statistics:

- gini – The Gini impurity of the node (explained below)
- samples – The number of samples in each node
- value – The breakdown of the number of samples of each target value in the node
    - For example, the leaf node on the left has 100 nodes labeled 0, and 0 nodes labeled 1
- A decision rule – The logic rule that is used for the following split
    - Samples that satisfy the rule go to the left child, while samples that are evaluated as False go to the right child
    - Leaf/terminal nodes do not have a decision rule, since they are the end of the tree

There are a number of ways one might measure the impurity of a decision tree, but we'll focus on Gini impurity and entropy here since they are easily implemented using sklearn

- They are specified in practice by setting criterion to either "gini" or 'entropy"

Assuming you have N target classes, the Gini impurity for class i of a node estimates the probability that a randomly chosen sample of class i from the node is incorrectly classified as not class i

The formula is $G_i = p_i(1 - p_i)$, where p_i is the proportion of samples in the node of class i, and the total Gini impurity is the sum G_i

$$I_G = \sum_{i-1}^{N} G_i = 1 - \sum_{i=1}^{N} p_i^2$$

The contribution made to entropy from class i, again assuming there are N target classes, is $H_i = -p_i \log(p_i)$, where again p_i is the proportion of samples in the node of class i, and the total entropy of the node is the sum of all the H_i

$$I_H = \sum_{i-1}^{N} H_i = -\sum_{i=1}^{N} p_i \log(p_i)$$

In most cases both measures are comparable, and since Gini impurity is faster to compute (the logarithm in entropy making it a bit more computationally taxing), it is a good default (and the default used by sklearn)

- It has been found that entropy leads to more balanced trees, though

When fitting a decision tree, sklearn uses the Classification and Regression Tree (or CART) algorithm

- Suppose your data set has n observations with m features, and for simplicity only 2 target classes

The algorithm starts with the root node, and then searches through each feature, k, until it finds a split point, t_k, that produces the purest subsets (weighted by the number of samples in each subset)

- I.e., it finds a split point t_k that minimizes

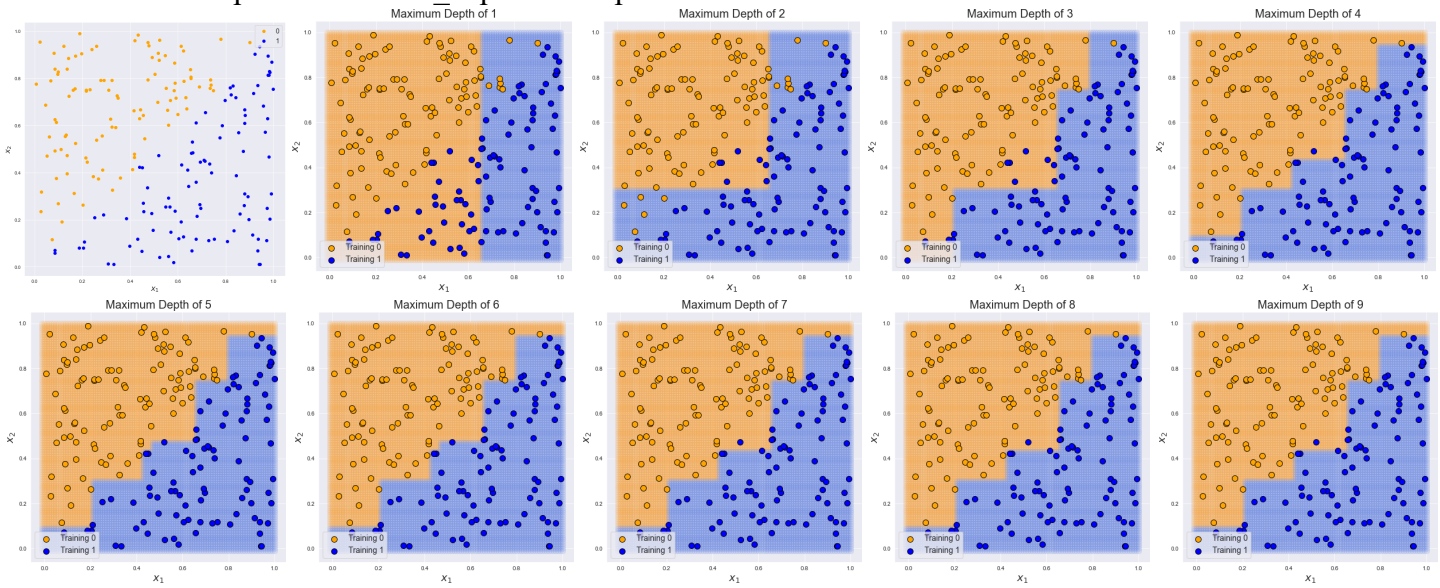
$$J(k, t_k) = \frac{n_{left}}{n} I_{left} + \frac{n_{right}}{n} I_{right}$$

- Once it finds the (k, t_k) pair with the smallest $J(k, t_k)$, it splits the data according to that decision split

The algorithm then repeats the entire process on each of the children nodes it just produced, and this continues until either it can no longer reduce the impurity by making a cut, or it reaches a stopping condition like:

- Reaching a maximum depth, controlled with max_depth
- Reaching a minimum number of samples in each node, controlled with min_samples_leaf
- Reaching a minimum weight to be in a node, controlled with min_weight_fraction_leaf
- Other possible stop conditions can be found in the DecisionTreeClassifier documentation

Here's at an example of how max_depth can impact a decision tree classifier:



Typically we'd choose our max_depth (or any other stop condition) using something like cross-validation

Decision Tree Advantages:

- Interpretability
  - As opposed to the "black box" moniker often used to describe machine learning, decision trees are known as a "white box" algorithm because you are able to entirely describe how they predict a data point using the logic tree
- Efficient
  - Fits are quick, and you require very little preprocessing of data prior to training
    - No need to scale data like we've needed to do for other algorithms

Decision Tree Disadvantages:

- Greediness
  - The algorithm is greedy meaning it may not create the optimal tree
  - If, for example, the best tree involves an initial suboptimal cut, the CART algorithm won't find it
- Overfitting
  - Decision trees are prone to overfitting the data, although you can control for this using regularization hyperparameters like max_depth and min_samples_split and cross-validation
- Orthogonal Boundaries
  - Because of the process of determining cut points, decision boundaries occur at right angles
  - This means that the decision tree will have difficulty capturing the boundary of classes divided by non-horizontal or non-vertical lines
  - This can be mitigated a bit with PCA
- Sensitivity
  - Trees are very sensitive to the training data, so that removing or adding a few points can greatly change the decision boundary produced by the algorithm
  - One way around this is to use an averaged algorithm, like a random forest that we'll discuss next

Sample Code:

```
# this will be used to plot the decision tree
from sklearn import tree
# import the actual classifier
from sklearn.tree import DecisionTreeClassifier

# make a decision tree object
tree_clf = DecisionTreeClassifier(criterion='gini')
# fit and then plot the decision tree
fig = tree_clf.fit(X, y)
tree.plot_tree(fig, filled = True)
plt.show()

# sample call using a max_depth stop condition
tree = DecisionTreeClassifier(max_depth = 5)
```

Nots on the above code:

- If all you want is to fit a decision tree, you can skip the "from sklearn import tree" bit above, as it's only needed for the plotting functionality here, and you don't need the full module just to fit a tree
- We haven't used it here, but there is also a decision tree regressor (found in DecisionTreeRegressor)

37. Random Forests I

`Lecture Notebooks/Supervised Learning/Classification/14. Random Forests I.ipynb`

A random forest classifier is comprised of numerous decision trees

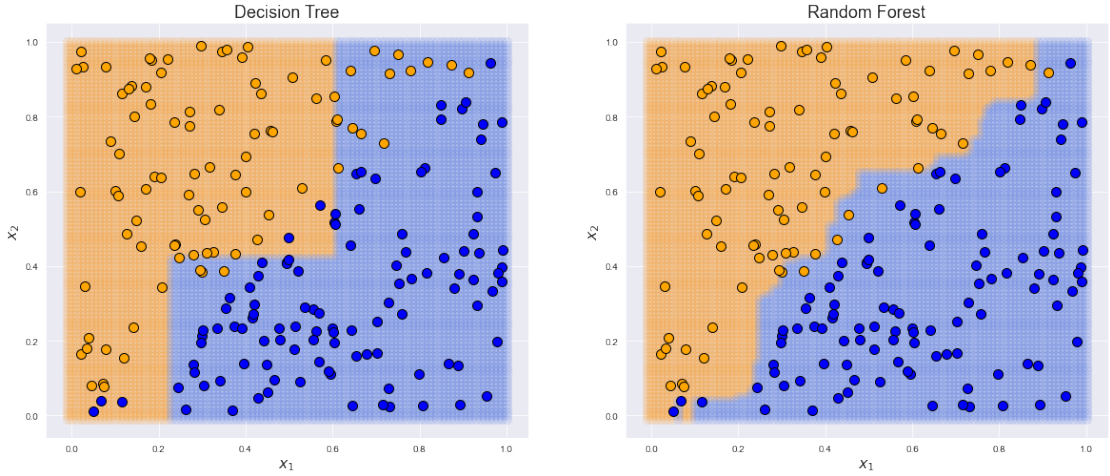
- The idea behind the algorithm is our first example of ensemble learning
- They are essentially a machine learning implementation of the wisdom of the crowd

The idea with ensemble methods is to build a number of different algorithms and then average their predictions into a "wiser" prediction

- In the case of random forests this means building many decision trees that are different (usually by some random perturbation) that are then used to produce a forest

In sklearn we can make a random forest classifier with RandomForestClassifier

Comparing a single decision tree with max_depth=2 to a random forest with the same max_depth=2 we can start to see the benefit of using an RFC over a single decision tree



There are several ways sklearn can go about building the decision trees needed for a random forest

- Something must be tweaked each time, otherwise you'd just repeatedly obtain the original decision tree

Random sampling with replacement (bagging)

- One way is to randomly sample training points with replacement from the data set, then train the algorithm on the randomly sampled set
  - So we build our set for training by repeatedly picking a point at random from our data set
- This process is more generally known as bagging, and we'll see it again when we touch on more general ensemble learning in future lessons
- Note that this is the default for sklearn's decision trees, and it can be controlled with the bootstrap option
  - Setting bootstrap=True uses bagging, while False trains each tree with the entire data set
- If your training set has n points, the algorithm by default randomly samples n points with replacement then trains a decision tree on it
  - This can be changed to be less than the entire dataset using the max_samples option
- The algorithm then trains n_estmiators different independent trees
  - The default value for n_estimators is 100, although this can easily be changed

Random sampling without replacement (pasting)

- An alternative to bagging is to randomly select data points without replacement from the training set
  - So if we randomly select a given point for inclusion in our test sample, it cannot be chosen again

        - As opposed to random sampling *with* replacement, where it *could* be chosen again
    - This is known as pasting
- We won't spend time on this one here since the random forests in sklearn don't currently support this option, but we'll revisit it in the future

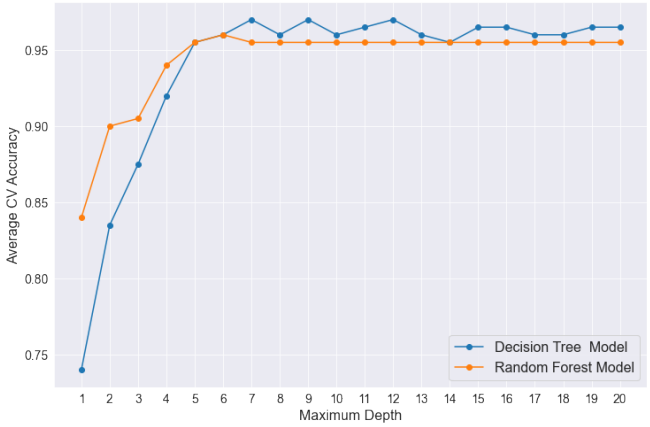
Randomly selecting predictors

- In addition to the ability to randomly sample data, every decision tree is built on a random sample of the features of the data
- This means that, unlike in a single decision tree where the best cut is chosen from all of the features at each step, we limit ourselves to which features we consider
- As with decision trees, you can control the maximum number of features considered in your model with the max_features option

Note that all of these options (bootstrap, max_samples, n_estimators, max_features, etc.) are hyperparameters

- Compared to the other algorithms we've examined thus, far random forests have the most hyperparameters to think about
- Depending on the settings you choose for the algorithm, you could wind up with vastly different predictions, so it's always important to think about why you choose a particular hyperparameter value

As before, cross-validation is a good way to tune hyperparameters like max_depth:

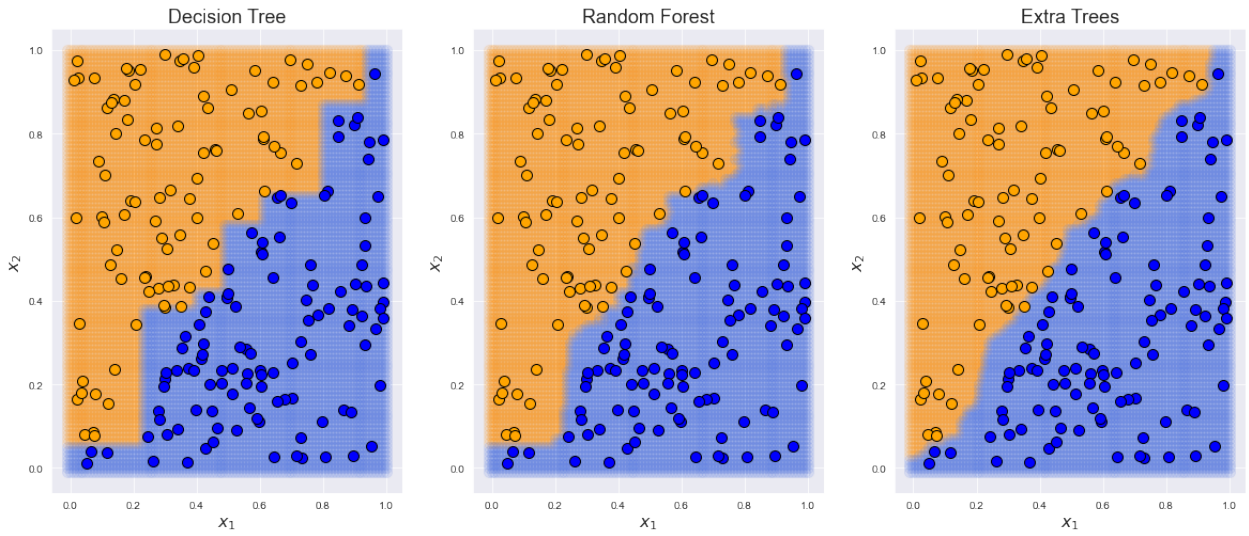


In this example we see that the best max_depth value for our RFC is about 6

- Note that although individual decision trees with large maximum depths can attain slightly higher CV accuracies, we would likely want to stick with the RFC given their propensity for overfitting
    - The oscillatory signatures we see for the decision tree at larger maximum depths indeed seems to suggest some overfitting

Extra-Trees

- This algorithm is just like a random forest, but in addition to randomly selecting a handful of features to optimize, it also randomly selects the cut-points rather than having the tree search for the optimal one
- This algorithm is faster random forests, but does tend to have a little more bias
    - Not necessarily a bad thing, since this naturally helps avoid overfitting
- Typically you'd want to build both classifiers and compare metrics via cross-validation to decide if an extra-trees classifier is better than a standard random forest for your application
- We can code this up in sklearn using ExtraTreesClassifier



An additional issue with tree-based algorithms in sklearn is that they don't handle categorical variables well

- Features are cast as floats in the fitting process, which works fine for continuous and binary predictors, but this is poorly suited to categorical variables
- When you one-hot encode a variable with many categories you end up with many sparse columns
    - The resulting sparsity virtually ensures that continuous variables are assigned higher feature importance
- A single level of a categorical variable must meet a very high bar in order to be selected for splitting early in the tree building
    - This can degrade predictive performance

It's not unreasonable to run into problems where this issue is pertinent, so what to do?

- You could try addressing your problem in R, since its 'rpart' package is better suited to tackling categorical variables
    - That said, R's 'randomForest' package may have limitations on the number of unique categories
- Alternatively you could look into other python packages that address this shortcoming
    - One example is h2o

Sample Code:

```
# import the random forest classifier
from sklearn.ensemble import RandomForestClassifier

# make a forest, fit the forest
forest = RandomForestClassifier(n_estimators=500, max_depth=2, random_state=614)
forest.fit(X,y)

# predict using the forest
forest_preds = forest.predict(X_pred)

# import ExtraTrees
from sklearn.ensemble import ExtraTreesClassifier

# make and fit an ExtraTrees classifier with max depth of 2
extra = ExtraTreesClassifier(n_estimators = 500, max_depth = 2, random_state = 2311, max_samples = 100)
extra.fit(X,y)
```

```
# get associated predictions
extra_preds = extra.predict(X_pred)
```

Notes on the above code:

- Because of the random elements in these two classifiers, it is again a good idea to include random_state values when you build them to aid in reproducibility
- All of the options we used for individual decision trees like min_samples_split can be used here as well

38. Random Forests II

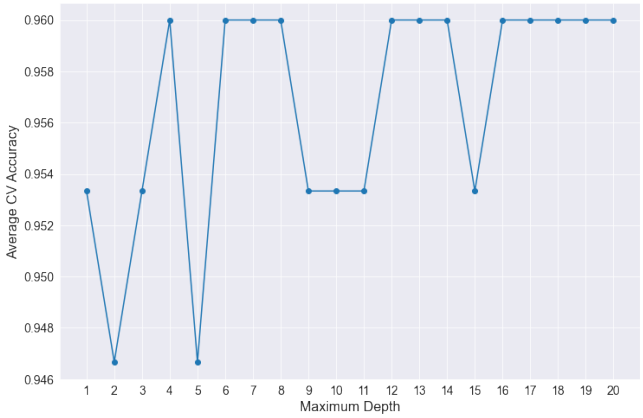
`Lecture Notebooks/Supervised Learning/Classification/15. Random Forests II.ipynb`

An additional benefit of the random forest algorithm is its ability to identify which features are important for classification

The sklearn algorithm measures feature importance in the following way:

1. For each feature it looks at every tree and identifies the nodes that use said feature to make a cut
2. It measures how much those cuts reduced impurity and averages that value over all the trees in the forest
3. After getting the average impurity reduction for each feature, sklearn scales the results so that the sum of all feature importances is equal to 1

As a practical example, let's again look at the iris data



	feature	importance_score
2	petal_length	0.450405
3	petal_width	0.435602
0	sepal_length	0.091570
1	sepal_width	0.022423

We begin by doing cross-validation to identify an optimal value for the maximum depth hyperparameter

- In this case, max_depth=4 looks like the best choice

We can then use the feature_importances_ attribute to identify which features are most important

- Here petal length/width are both quite significant, while sepal length/width are relatively unimportant

Sample Code:

```
# import the RFC package
from sklearn.ensemble import RandomForestClassifier
# make and fit a random forest
forest = RandomForestClassifier(n_estimators=500, max_depth=4)
forest.fit(X_train, y_train)

# demonstrate feature importances
forest.feature_importances_

# make it a more readable dataframe
score_df = pd.DataFrame({'feature':X_train.columns, 'importance_score': forest.feature_importances_})
score_df.sort_values('importance_score',ascending=False)
```

Notes on the above code:

- In this example we had previously done cross-validation to identify an optimal value for max_depth of 4
- The same feature_importances functionality can be applied to extra-trees classifiers made from sklearn's ExtraTreesClassifiers package

39. Ensemble Learning I – Voter Models

`Lecture Notebooks/Supervised Learning/6. Voter Models.ipynb`

Ensemble models can be used for both regression and classification tasks

- For example, while we introduced random forests as a classification method, they can easily be used for regression, all you need to do is create a forest of decision tree regressors
- In this lesson we'll focus on voter models for classification, but these too can work for regression

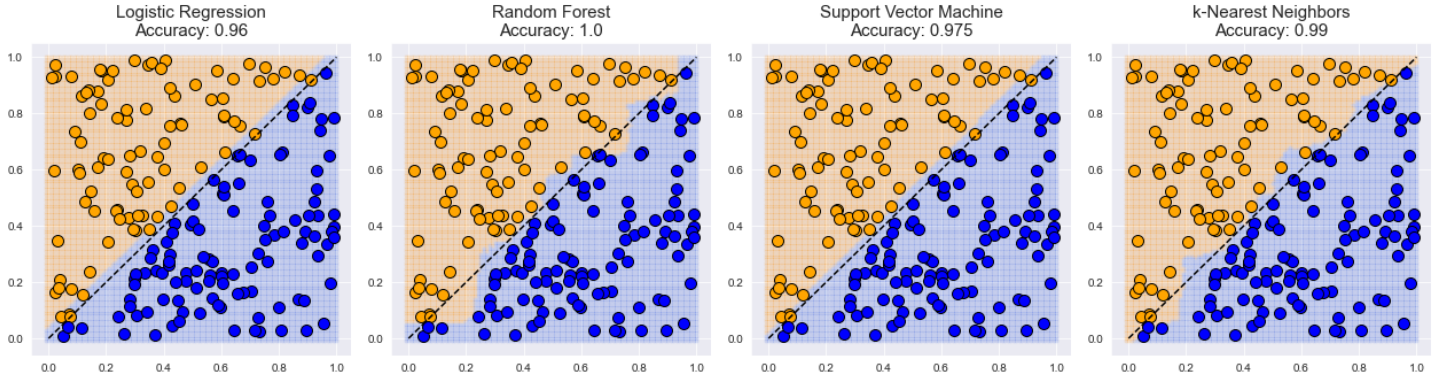
Let's say that you have a few different classifiers that you think are pretty good, for instance a logistic regression model, a knn model, a support vector machine, and a random forest model

- A voting classifier is one that looks at how each of your classifiers decides to classify a point and goes with the decision that receives the most "votes"

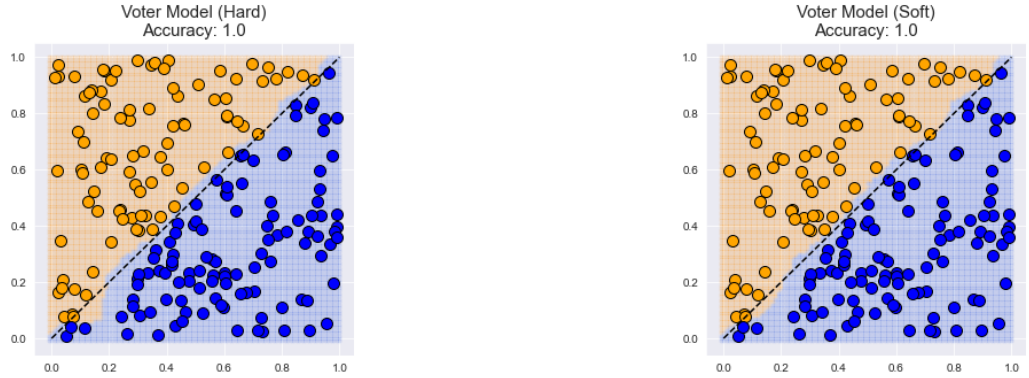
We can implement voting models in sklearn using the VotingClassifier object

- Let's look at example voting classifiers built around a logistic regression classifier, a random forest classifier, a support vector machine classifier, and a *k*-nearest neighbors classifier

Component models:



Resultant voter models:



Note that we coded up the above example first using hard voting (by including the argument voting="hard")

- With this method, the prediction is decided according to the majority vote of the individual classifiers
- For example, if 3 out of 4 possible classifiers classify the observation as a 1, then the voter model classifies it as a 1

The other option (used for right plot) is to use soft voting (by including the argument voting="soft")

- For this type of voting classifier predictions are chosen according to the probabilities assigned by each of the constituent classifiers
- For each observation the soft voter assigns the class, c, for which $\sum_{j=1}^{V} P_j(y_i = c|X_i)$ is largest, where P_j denotes the probability according to voting classifier j of V possible classifiers

- This does a better job handling the notch of sparsely sampled data in the lower left of the above example

Note that you can also perform weighted voting of the classifiers with the weights argument

- This would assign some classifiers more pull in deciding the predicted class than others

Voter models can also be made using an ensemble of independent regression models

- The voter regression model takes the average (or weighted average) of all of the regression models fed into it to make a prediction
- Implemented using sklearn's VotingRegressor object
- Note that this does not mean that you should build several linear regression models with slightly different features and feed them into a voter model
    - This applies to voter model classifiers as well!
- Instead you should build a handful of *unique* regression models, for example using a linear regression model, a KNN regression model, a support vector regressor, and a random forest regressor
    - We introduced these methods primarily as classifiers, but they can all also be used for regression

Sample Code:

```
# import base classifiers
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier

# now import the voting classifier
from sklearn.ensemble import VotingClassifier

# make base models
log = LogisticRegression()
knn = KNeighborsClassifier(5)
svm = SVC(kernel='linear', C=1, probability=True)
rf = RandomForestClassifier(max_depth=5)

# make the voting classifier
voting = VotingClassifier( [('log', LogisticRegression()),
    ('knn', KNeighborsClassifier(5)), ('svm', SVC(kernel='linear', C=1, probability=True)),
    ('rf', RandomForestClassifier(max_depth=5))], voting='hard' )

# fit them all with a for loop
for name,clf in ( ["log_clf",log], ["rf_clf",rf],  ["svm_clf",svm], ["knn_clf",knn], ["voting_clf",voting] ):
# fit the specific model
 clf.fit(X, y)
   # get predictions
   y_pred = clf.predict(X)
```

Notes on the above code:

- Note that the voting classifier syntax is similar to that we saw earlier for pipelines

40. Ensemble Learning II – Bagging and Pasting

`Lecture Notebooks/Supervised Learning/7. Bagging and Pasting.ipynb`

Bagging and pasting work by training the same kind of classifier on "different" data sets

- Different data sets are obtained by randomly sampling our training data to simulate a new draw of training data
- When that sampling is done with replacement, it is known as bagging
  - This is short for "bootstrap aggregating" and is sometimes referred to as bootstrap sampling
  - I.e., grab 1 M&M out of your hypothetical bag of them, record its color, and then put it back, repeating this process 10 times
- When sampling is performed without replacement, it is called pasting
  - I.e., grab 10 M&Ms out of your bag at once to use as your simulated sample

Using syntax similar to the VotingClassifier package, we can use sklearn's BaggingClasifier package to build both pasting and bagging models

Note that unlike in the voter model, where we wanted to use a number of fundamentally different classifiers, here we will be using an ensemble of similar classifiers trained on the slightly different sample data sets produced through either bagging or pasting

- This is similar to the idea behind taking an ensemble of decision trees to build a random forest classifier

As an example, we can build bagging and pasting models using an ensemble of KNN classifiers and compare against that of a standard individual KNN classifier

Looking at the plots above, we see that for this data set the bagging classifier seems to do a bit better than the pasting classifier

- They both are a rather significant improvement over the individual KNN classifier, which almost entirely misses blue classifications for $x_1 < 0.2$

Note that when using these kind of models you'll still want to optimize hyperparameters for the base model (like the number of nearest neighbors) using cross-validation as covered previously

- Also good to compare between bagging and pasting, as these are hyperparameters for this model

Just like with voting models, bagging and pasting can be implemented with regression models as well, where the prediction for a particular set of features X^* is taken as the average of all the bagging base model predictions

- In sklearn this is implemented with BaggingRegressor

Sample Code:

```
# import model objects
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import BaggingClassifier

# make bagging and pasting classifiers
bag = BaggingClassifier(base_estimator = KNeighborsClassifier(10),
                        n_estimators = 100,
                        max_samples = 100,
                        bootstrap = True,
                        random_state = 7556)
paste = BaggingClassifier(base_estimator = KNeighborsClassifier(10),
                        n_estimators = 100,
                        max_samples = 100,
                        bootstrap = False,
                        random_state = 892)

# we'll compare it to a single knn
knn = KNeighborsClassifier(10)

# fit individual knn and check accuracy
knn.fit(X,y)
y_pred = knn.predict(X)
knn_acc = sum(y == y_pred)/len(y_pred)

# fit bagged data and check accuracy
bag.fit(X,y)
y_pred = bag.predict(X)
bag_acc = sum(y == y_pred)/len(y_pred)

# fit paste data and check accuracy
paste.fit(X,y)
y_pred = paste.predict(X)
paste_acc = sum(y == y_pred)/len(y_pred)
```

Notes on the above code:

- In BaggingClassifier, if bootstrap=True we're using bagging, while bootstrap=False uses pasting
- As in most applications, random states aren't strictly necessary to use, but are helpful for reproducibility

41. Ensemble Learning III – AdaBoost

`Lecture Notebooks/Supervised Learning/8. AdaBoost.ipynb`

Boosting is a very powerful algorithm, utilized in a number of winning data science competition entries

- The theory behind the algorithm is based on the concepts of weak learners and strong learners from the subfield in Statistical Learning on PAC learnability
    - Here PAC is short for "Probably Approximately Correct"

A statistical learning algorithm is referred to as weak learner if it does slightly better than random guessing, while a strong learner can be made arbitrarily close to the true relationship

- In practice, a common weak learner algorithm is a single-layered decision tree, or a "decision stump"

Making a weak learner is much easier than producing a strong learner in general

- However, it has been shown that if a problem is weak learnable (meaning that a weak learner exists) then it is strong learnable (meaning that a strong learner exists)
    - Granted, the fact that a strong learner exists does not mean it is easy to identify

The idea for boosting is that we combine an ensemble of weak learners to produce a strong learner

- Here we will focus specifically on the AdaBoost algorithm developed by Freund and Schapire (1997)
- Like our other ensemble learners, boosting can be used for either regression or classification, but here we will focus on its application to classification problems

AdaBoost, which is short for adaptive boosting, trains a series of weak learners, with each subsequent classifier paying more attention to the training instances that were misclassified by its predecessors

- For a weak learner j, the weights on each training sample are determined by the performance of weak learner $j-1$
- After training W total weak learners, a final prediction is made by performing a weighted vote among all W weak voters
    - In classification problems, voting weight is determined by the learner's accuracy

Let $y^{(i)}$ denote the class of observation i, $\hat{y}_j^{(i)}$ denote the prediction on observation i of weak learner j, and $w^{(i)}$ denote the current weight assigned to observation i

After the j^{th} weak learner is trained, that learner's weighted error rate is calculated as

$$r_j = \frac{\sum_{i=1}^{n} w^{(i)} 1_{\left\{\hat{y}_j^{(i)} \neq y^{(i)}\right\}}}{\sum_{i=1}^{n} w^{(i)}} = 1 - \text{weighted accuracy}$$

So r_j is large when the j^{th} weak learner performs poorly, and r_j is small when it performs well

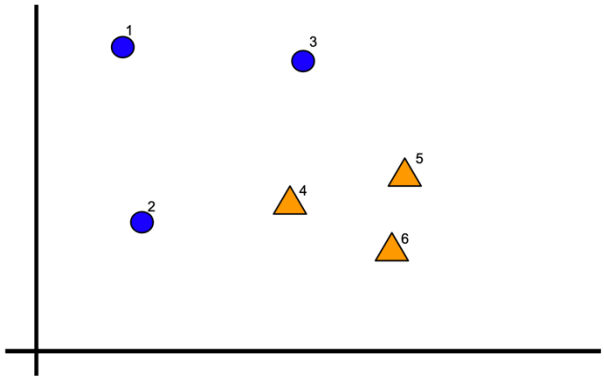
The next step is to compute the weight assigned to the weak learner itself, $\alpha_j = \eta \log\left({}^{1-r_j}/_{r_j}\right)$, where η is the learning rate of the algorithm, a hyperparameter you must set prior to fitting

- Note that α_j is small when r_j is large (the j^{th} weak learner is good), and large when r_j is small (it's bad)

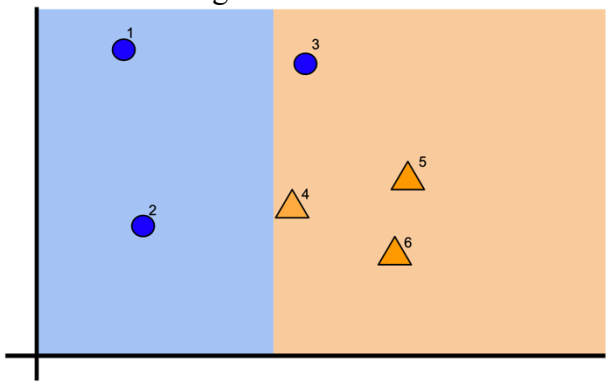
Finally, we update the training sample weights for weak learner $j+1$, increasing the weight on incorrectly predicted samples:

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp\left(a_j\right) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

That's all a bit abstract, so let's look at training an AdaBoost classifier on this toy data set:



For the first weak learner, each observation is given a uniform weight $w^{(1)}, \ldots, w^{(6)} = 1/6$ and using those weights the first weak learner yields the following decision rule:



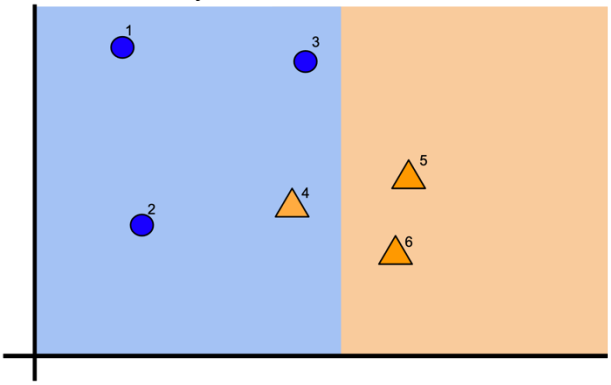
AdaBoost now calculates in order r_1, α_1, and new weights w (for simplicity here we'll take $\eta = 1$)

$$r_1 = \frac{0+0+1/6+0+0+0}{1/6+1/6+1/6+1/6+1/6+1/6} = \frac{1}{6}, \qquad \alpha_1 = \log\left(\frac{1-1/6}{1/6}\right) = \log(5)$$

Because they were correctly classified, w_i for most points remains $1/6$, but since point 3 was misclassified

$$w^{(3)} \to \frac{1}{6}\exp(\log(5)) = \frac{5}{6}, \qquad \text{and} \qquad w = (1/6, 1/6, 5/6, 1/6, 1/6, 1/6)$$

These new weights are then used when training the second weak learner which, due to the increased weight on observation 3 produces a new decision boundary:



AdaBoost now again calculates r_2, α_2, and updates the weights

$$r_2 = \frac{0+0+0+1/6+0+0}{1/6+1/6+5/6+1/6+1/6+1/6} = \frac{1}{10}, \qquad \alpha_2 = \log\left(\frac{1-1/10}{1/10}\right) = \log(9)$$

Here observation 4 is the only misclassification, so we update its previous weight of $1/6$ to find

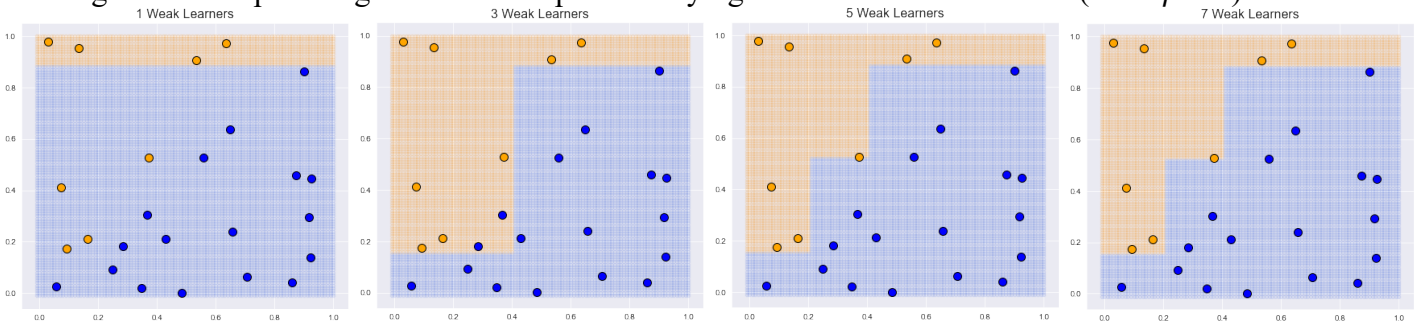
$$w^{(4)} \to \frac{1}{6}\exp(\log(9)) = \frac{3}{2}, \qquad \text{and} \qquad w = (1/6, 1/6, 5/6, 3/2, 1/6, 1/6)$$

If we were to stop here, predictions would be made using a weighted vote where weak learner 1 has votes worth $\alpha_1 = \log(5)$ and weak learner 2 has votes worth $\alpha_2 = \log(9)$

- The number of weak learners to use before stopping is a hyperparameter, one set by n_estimators

We can implement this algorithm in sklearn using the AdaBoostClassifier package

Looking at an example using decision stumps and varying the number of estimators (with $\eta = 1$) we see:



As we increase the number of weak learners, we tend to overfit the training data

- This can be mitigated by not using too many estimators
    - Where, as always, the "correct" number of estimators depends on your application and can be chosen with something like cross-validation

Sample Code:

```
# import AdaBoost
from sklearn.ensemble import AdaBoostClassifier
# import the base classifier needed for decision stumps
from sklearn.tree import DecisionTreeClassifier

for i in [1,3,5,9]:
ada_clf = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1),
                             n_estimators=i, algorithm = 'SAMME.R',
                             random_state=123)
  # fit the classifier
  ada_clf.fit(X, y)
  # make predictions from fit classifier
  preds = ada_clf.predict(X_pred)
```

Notes on the above code:

- The learning rate η takes on a default value of 1.0 if not defined in the AdaBoost object
- Setting max_depth=1 for the decision tree forces it to be a decision stump
- When declaring the AdaBoost algorithm, using "SAMME.R" is important since it allows our classifier to give us probabilities
    - The other option, "SAMME", does not return probabilities

42. Ensemble Learning IV – Gradient Boosting

`Lecture Notebooks/Supervised Learning/9. Gradient Boosting.ipynb`

Gradient boosting is another boosting technique in which we iteratively build an ensemble of weak learners with the hope of creating a strong learner

- The approach is to directly train weak learner $j + 1$ to model weak learner j's errors
- We'll focus on regression here, but as with most other algorithms we've looked at it can also be used for classification

Recall that in regression we try to model a quantitative variable y using m features contained in a matrix X

Gradient boosting in the context of regression works like so:

Step 1)

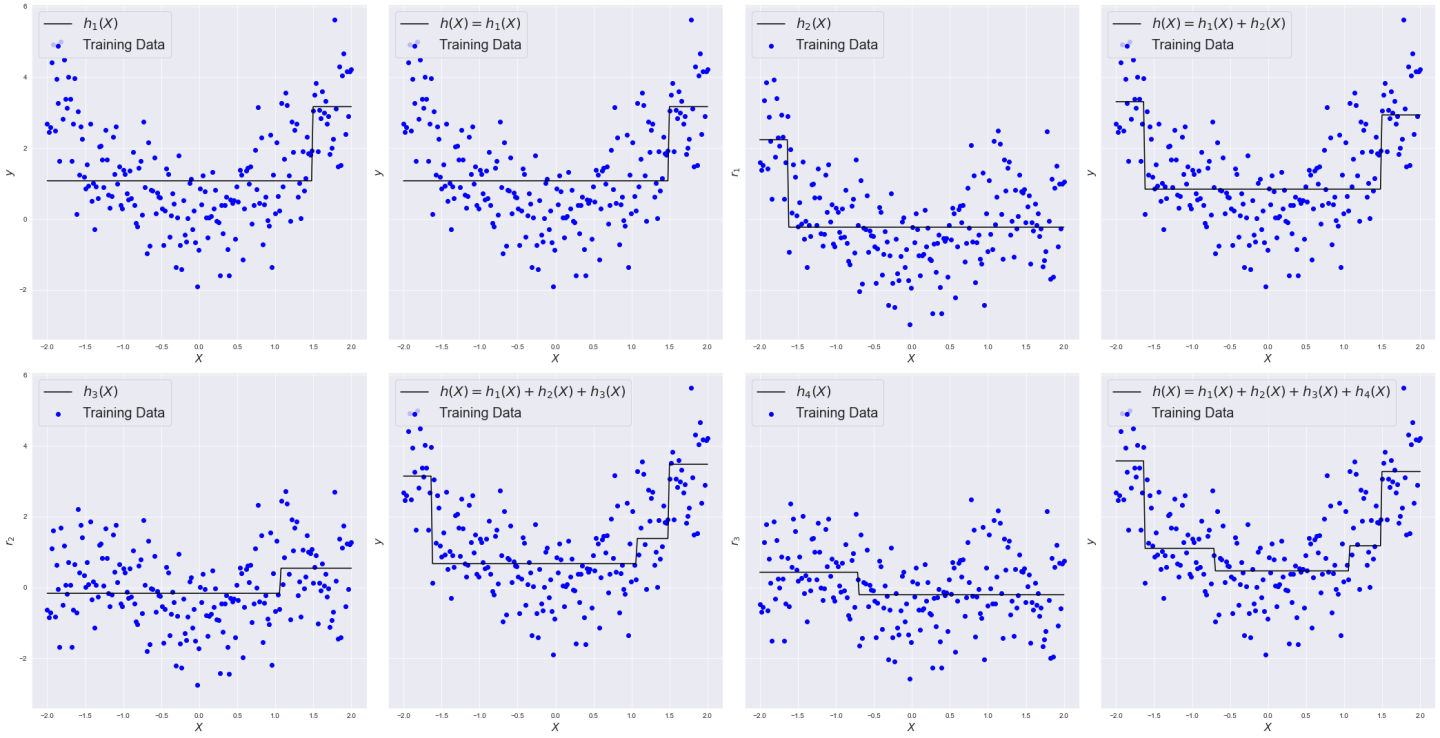
- Train a weak learner regression algorithm (say, a decision stump regressor) to predict y
  - This is weak learner 1
- Calculate the residuals $r_1 = y - h_1(X)$, where $h_1(X) = \hat{y}$ is the prediction of weak learner 1

Step j)

- Train a weak learner on the residuals from step $j - 1$
  - Let $h_j(X) = \hat{r}_{j-1}$ denote the j^{th} weak learner's estimate of the residuals
- Calculate the residuals for this weak learner, $r_j = r_{j-1} - h_j(X)$
- Repeat, stopping when $j + 1 = J$
  - J is a predetermined stopping point (and thus a hyperparameter for this algorithm)

The prediction for y at step j is found as $h(X) = h_1(X) + h_2(X) + \cdots + h_j(X)$, adding up the predictions from each of the weak learners fit so far

In practice this looks something like so



For regression, we implement gradient boosting in sklearn with the GradientBoostingRegressor object

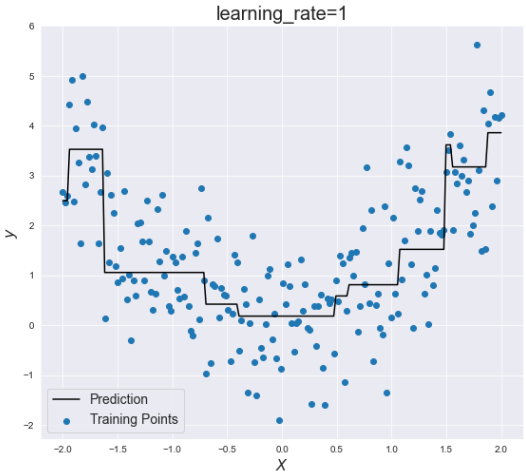
- This follows the same sequence as the above example, training a series of decision tree regressors (with default max_depth=3)

- The number of trees trained is determined by n_estimators (with a default value of 100)

The learning rate determines how much weight each weak learner receives in the final prediction and offers one way to control overfitting

- Note, however, that is typically preferable to use a small learning rate and then train more trees instead of using a large learning rate
- This is likely why sklearn's default value for learning_rate is 0.1

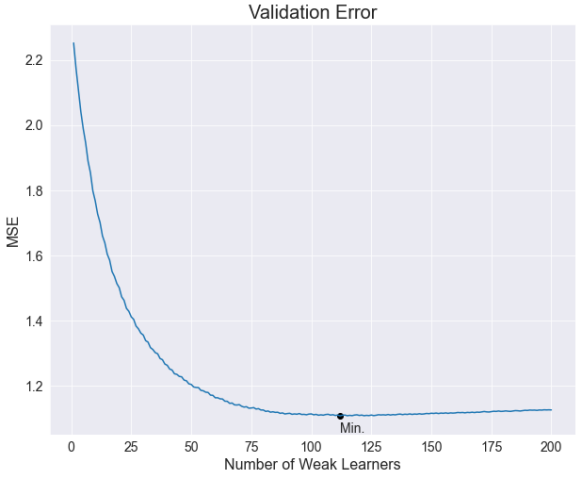
The impact of using a small or large learning rate

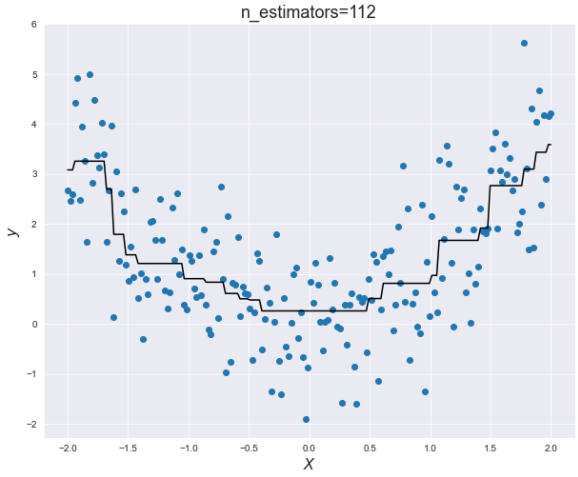


A second way to control over/underfitting this algorithm is by controlling the number of weak learners you train

One way to find this that may be preferable to cross-validation is to use a validation set

- You track the validation set error as you train additional weak learners, and then you pick the number of weak learners that had the lowest validation set error
  - We can implement this in sklearn using the staged_predict method, which returns an iterator over the predictions made by the booster at each level (with one tree, with two trees, etc.)
- In this example, 112 weak learners has the lowest validation error and indeed seems to be a good fit





We can also implement what is known as early stopping, where we stop training additional layers when it appears that we have reached a minimum

- To do so in sklearn, we use the warm_start argument
  - This forces sklearn to keep older layers when the fit method is called

- In practice, we could code up a loop that stops if, after attaining what seems to be a minimum validation error, the next 10 weak learners produced worse error metrics
- This saves time compared to training many trees and then looking retrospectively to identify the best one
- This approach it is somewhat susceptible to getting stuck in a local minimum of your loss function
    - You can help mitigate this by doing something like stochastic training (where you do a random jump in the number of additional layers) to allow the algorithm to escape from local minima
    - Although standard early stopping should be fine for gradient boosting and the MSE

So why is it called "gradient" boosting, anyway?

Denoting our prediction for y as $\hat{y} = H_j(X) = \sum_{k=1}^{j} h_k(X)$ for step j, then for step $j+1$ we have

$$y \approx H_{j+1}(X) = H_j(X) + h_{j+1}(X) \rightarrow h_{j+1}(X) \approx y - H_j(X)$$

Recall that for regression problems we have typically tried to minimize the MSE of the estimate, which at the $j+1$ step we can denote as:

$$\frac{1}{n}\Big(y - H_j(X)\Big)^2$$

Taking the negative gradient of this with respect to the estimate H_j gives

$$\frac{2}{n}\Big(y - H_j(X)\Big) \approx \frac{2}{n} h_{j+1}(X)$$

Thus we see that gradient boosting is roughly speaking a gradient descent algorithm

Sample Code:

```
# first import GradientBoosting
from sklearn.ensemble import GradientBoostingRegressor
# also import mse
from sklearn.metrics import mean_squared_error

# make the model object
n_trees = 200
gb = GradientBoostingRegressor(max_depth=1,n_estimators=n_trees)
# fit the booster
gb.fit(X.reshape(-1,1), y)

# use a list comprehension and staged_predict to get validation errors for 1 to 200 n_estimators
mses = [mean_squared_error(y_val, pred) for pred in gb.staged_predict(X_val.reshape(-1,1))]

# now make a model using the lowest validation mse
best_num = range(1,n_trees+1)[np.argmin(mses)]
gb = GradientBoostingRegressor(max_depth=1, n_estimators=best_num)
gb.fit(X.reshape(-1,1), y)
```

Notes on the above code:

- This shows a general validation set optimization approach for identifying the best number of estimators
    - If the explanation for early stopping above isn't clear, see the associated notebook for a coded up example of utilizing that method
- As before, setting max_depth=1 tells the algorithm to use only decision stumps

43. Ensemble Learning V – XGBoost

`Lecture Notebooks/Supervised Learning/10. XGBoost.ipynb`

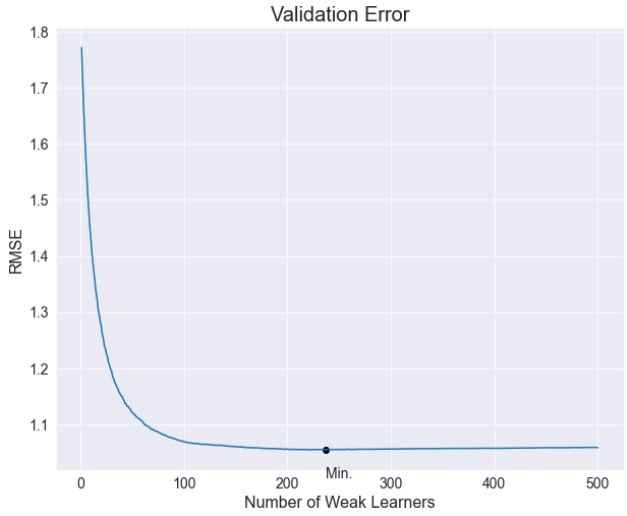
In the previous lesson we learned about gradient boosting was and how to implement it using sklearn's GradientBoostingRegressor model object

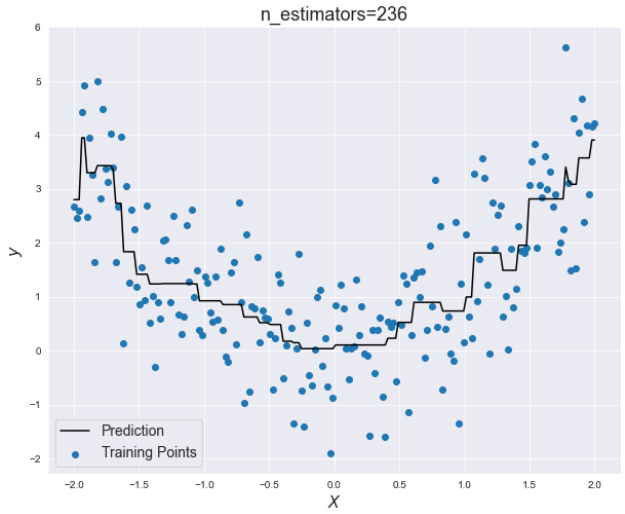
- Recall that this technique is a boosting approach where we iteratively train weak learners by training on the previous learner's residuals

Beyond sklearn, another popular package for gradient boosting is XGBoost (which stands for eXtreme Gradient Boosting)

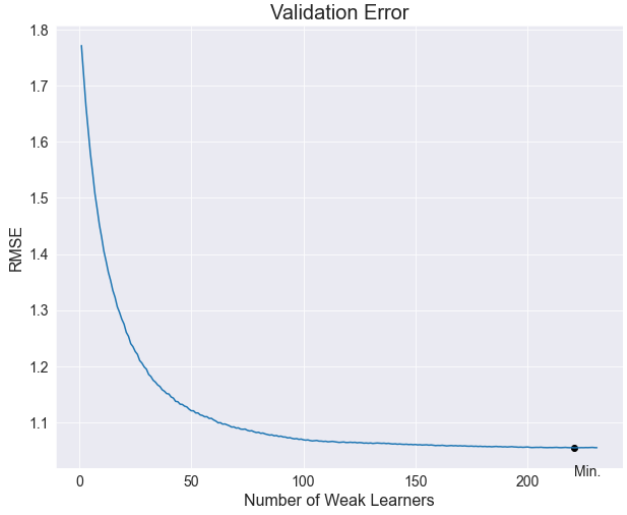
- XGBoost's code for fitting gradient boosting models is much faster and tends to perform better than sklearn's implementation
    - It even offers the capability for your model to be trained in parallel, which sklearn does not currently offer for gradient boosting
- Gradient boosting regression is implemented using XGBRegressor, and classifying with XGBClassifier
- Note that this package isn't as ubiquitous as sklearn, so you may need to install it on your machine if you haven't used it previously (and you can usually do so using pip/conda install)

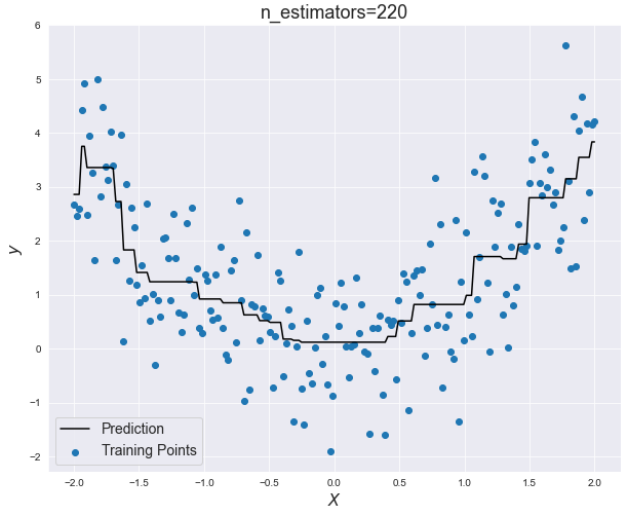
The XGBoost package offers a nice way to track validation set performance for optimization:





Additionally, we can implement early stopping in XGBoost without having to write our own clunky loop:





Sample Code:

```
# import XGBoost package
import xgboost

# FIRST DEMONSTRATE VALIDATION SET OPTIMIZATION WITH XGBOOST
# make a regressor object to
xgb_reg = xgboost.XGBRegressor(n_estimators=500, max_depth=1, learning_rate = 0.1)
# fit the model, including an eval_set
xgb_reg.fit(X.reshape(-1,1), y, eval_set=[(X_val.reshape(-1,1), y_val)])

# get number of estimators that has the best RMSE in the total validation set comparison
best_num = np.argmin(xgb_reg.evals_result()['validation_0']['rmse'])
# use this to build the optimal regressor
xgb_optimal = xgboost.XGBRegressor(n_estimators=best_num, max_depth=1, learning_rate = 0.1)
xgb_optimal.fit(X.reshape(-1,1),y)

# NOW DEMONSTRATE EARLY STOP OPTIMIZATION WITH XGBOOST
# start with the same regressor object as before
xgb_reg = xgboost.XGBRegressor(n_estimators = 500, max_depth = 1, learning_rate = 0.1)
# now use early_stopping_rounds
xgb_reg.fit(X.reshape(-1,1), y, eval_set=[(X_val.reshape(-1,1), y_val)], early_stopping_rounds=10)

# get number of estimators that has the best RMSE in the early stop set
best_num = np.argmin(xgb_reg.evals_result()['validation_0']['rmse'])
# use this to build the optimal regressor
xgb_optimal = xgboost.XGBRegressor(n_estimators=best_num, max_depth=1, learning_rate = 0.1)
xgb_optimal.fit(X.reshape(-1,1),y)
```

Notes on the above code:

- In general XGBoost is designed to feel similar to sklearn packages, so we can conveniently retain most of the syntax we've been using in prior lessons
- Setting eval_set lets us more conveniently store the validation set metrics for our series of regressors
    - We can then access these metrics with evals_result
- Here we are fitting the same toy data used in the prior lesson, and since this data is 1-D we need to again use our old friend reshape(-1,1)

44. Ensemble Learning Summary

`Lecture Notebooks/Supervised Learning/11. Ensemble Learning Summary.ipynb`

In review, we have learned about the following ensemble learning methods:

1. Voter Models
2. Bagging/Pasting Models
    a. Random Forests
3. Boosting Models
    a. AdaBoost
    b. Gradient Boosting
        i. XGBoost

Training in Parallel

- One nice perk of voting models and baggers/pasters is that their constituent models can be trained in parallel (at the same time)
- In contrast, the AdaBoost and Gradient Boosting algorithms need to wait for the weak learner at step j to be trained prior to training the weak learner at step $j + 1$
- This means that, in general, voting models and baggers/pasters may be faster than the two boosting algorithms that we have learned about (assuming the constituent models are relatively quick to train)

Constituent Models

- Building off of our last point, while you cannot parallelize the constituent models of the boosting algorithms, because they are weak learners they are quick and easy to train (recall our decision stumps)
- By contrast, voting and bagging/pasting models will likely have more complex constituent models, which take longer to train

Importance of Model Independence

- Voting models work best when they consist of more or less independent techniques
- If all of the models in your voting method tend to make the same mistakes, then the voting model is unlikely to significantly outperform its constituent models
- This can limit their usage if, for example, you are unable to train a variety of different models that perform well

45. Perceptrons

`Lecture Notebooks/Supervised Learning/Neural Networks/1. Perceptrons.ipynb`

Neural networks are a technique that that loosely tries to mimic the network of neurons that make up brains

- The idea being that we're trying to create learning algorithms that copy in some very loose sense how humans learn
    - Although in practice neural networks work quite differently from the human brain
- Introduced by Rosenblatt in 1960, perceptrons are the fundamental building block of neural networks

Note that, while we're going to discuss this in the setting of a classification problem, neural nets can be applied to a variety of settings

Suppose we have n observations of m features stored in m different n by 1 column vectors $X_1, X_2, \dots, X_m$ with $X = (X_1|X_2| \dots |X_m)$ and we want to predict some target y

- Note that we could include a column of 1s, but we'll leave it out of this formulation and come back to it

Let σ be some nonlinear function from $\mathbb{R} \to \mathbb{R}$

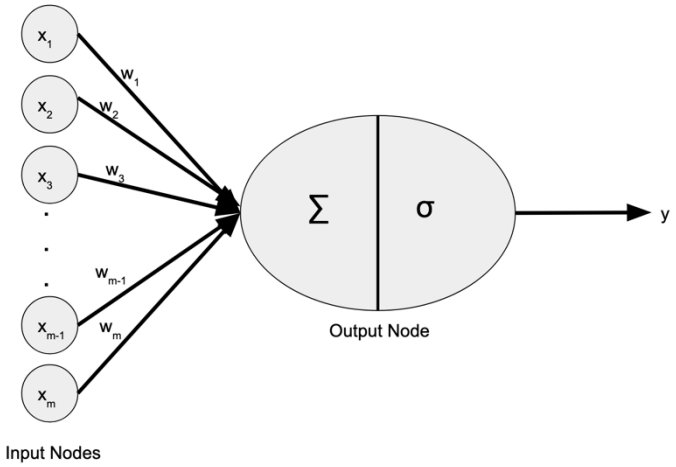
- In the language of neural networks we call σ an activation function
- For classification we take $\sigma = \text{sgn}$, the sign function
    - This states simply that $\sigma(x) = 1$ if $x > 0$, and $\sigma(x) = -1$ if $x < 0$

Perceptrons make an estimate of y like so

$$\hat{y} = \sigma(w_1 X_1 + w_2 X_2 + \cdots + w_m X_m) = \sigma(Xw),$$

where, in a potential abuse of notation, we have taken $\sigma(Xw)$ to mean σ applied to each of the n entries of Xw, and $w = (w_1, w_2, \dots, w_m)^T$

Letting $x = (x_1, x_2, \dots, x_m)$ denote a single observation, I can draw the architecture of the neural network as



The column of nodes are the inputs into the perceptron, and the output node has both Σ and σ because this is where our weighted sum (Σ) and nonlinear transformation (σ) occur

- Note that, while we did not include a bias term (adding a constant to Xw), this can be easily done and is just left out here for simplicity

Initial weights are selected randomly, and from there you then use a single data point from the training set, $X^{(i)}$, and you calculate the error $y^{(i)} - \hat{y}^{(i)}$

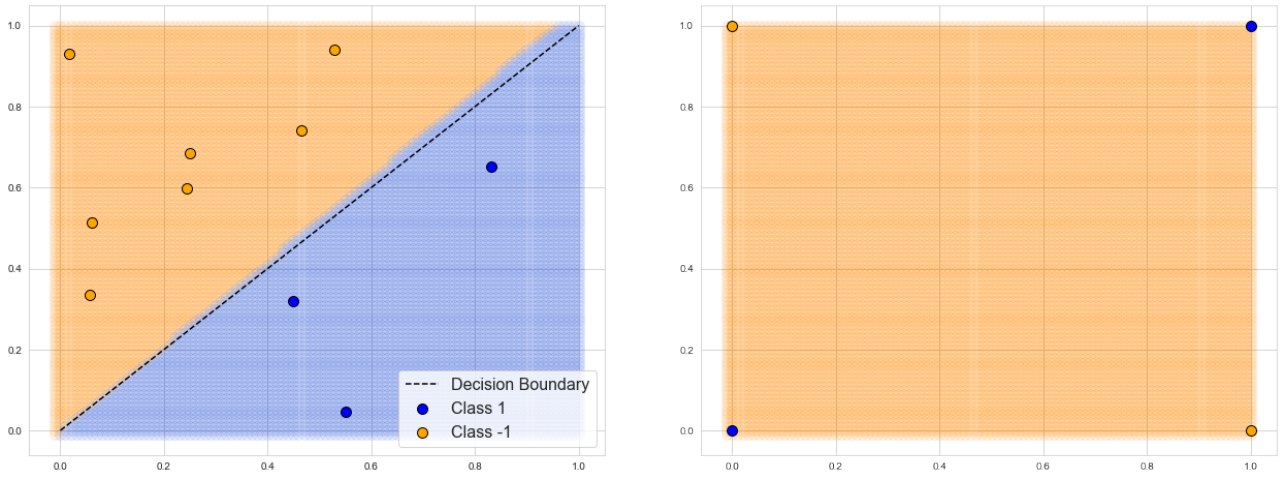
You then update w to be $w_{update} = w_{current} + \alpha\left(y^{(i)} - \hat{y}^{(i)}\right)X^{(i)}$, where α is the network's learning rate

The perceptron will cycle through all of the training points and continue to adjust the weights until it converges to a final weight vector w

- Each cycle through the training set is called an epoch
- Typically the training points are chosen at random without replacement
- Note that this can be performed with small batches of training points, at which point the batches are chosen randomly without replacement
- Note also that this algorithm can be rewritten to work in parallel

Perceptrons can be implemented in sklearn using the Perceptron package

Looking at a couple toy examples, we can quickly see the perceptron's Achilles' heel



Clearly this method doesn't work very well for the second example, and indeed, a single perceptron is not capable of separating data sets that are not linearly separable

- This limitation greatly reduced interest in perceptrons back in the 1950s and 60s
- If your data *is* linearly separable, however, there is a proof that guarantees the perceptron will converge, as well as an upper bound on the number of epochs it must endure to get there

The linear limitation is a major hindrance, though, so we'll move on to learning about more complicated neural network architecture in our next lessons

Sample Code:

```
# import the model
from sklearn.linear_model import Perceptron

# make and fit a perceptron object
perc = Perceptron()
perc.fit(X, y)

# get predictions from model for your test set
preds = perc.predict(X_test)
```

Notes on the above code:

- Given the limitations of individual perceptrons, you probably won't use this in practice since better alternatives exist for linearly separable classification problems

46. The MNIST Data Set

`Lecture Notebooks/Supervised Learning/Neural Networks/2. The MNIST Data Set.ipynb`

Let's take a look at the MNIST data set, since we'll be using it extensively in coming lessons

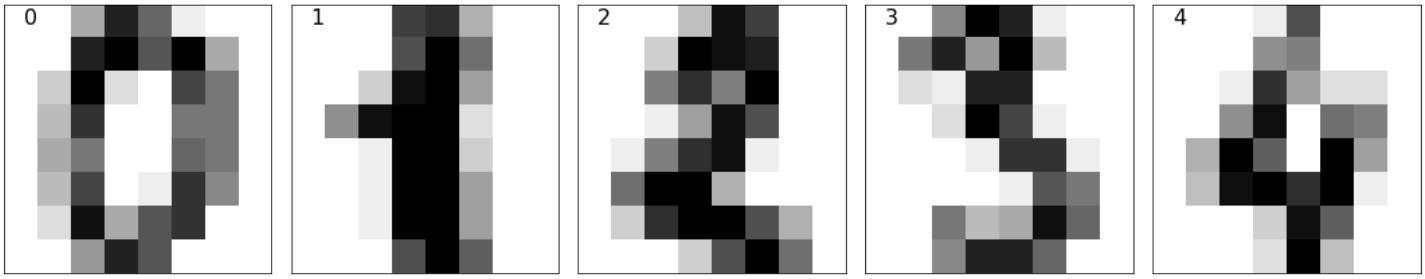
- We won't look at it during these lessons, but keras also includes a similar data set called the "MNIST of Fashion" that has pre-labeled images of 70,000 clothing items
    - This might be fun to explore if you're interested in more practice with neural networks

The MNIST (Modified National Institute of Standards and Technology) data set is a collection of pixelated images of handwritten digits (the counting numbers from 0-9)

- Each image is broken into a grid of pixels of grayscale values which measure the intensity of the handwriting within that pixel
- Each pixel's value ranges from 0 (no marking) to 255 (darkest marking)
- The original data set contained 60,000 training images and 10,000 test images

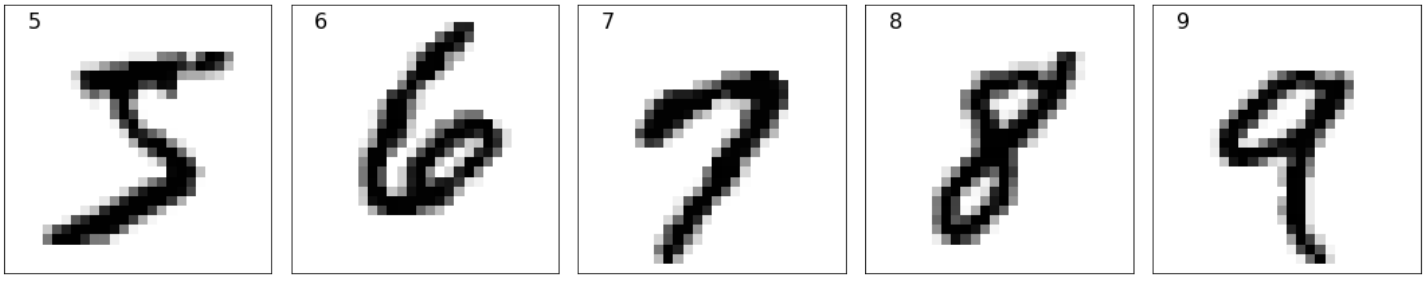
The sklearn version of this data set is a small sample of the original data with lower resolution images

- It can be loaded with the load_digits function from the datasets module
- This version of the data contains 1,797 different 8x8 images
- Not a high quality version of the data, but it works well enough for demonstrating and testing machine learning algorithms



The other version of this data set that we will use can be found in the keras package

- In keras the MNIST data can be loaded with mnist in the datasets module
    - This is the full version of the data set, with 60,000 training observations and 10,000 test observations of 28x28 pixel images
- keras is a python package built for making neural network models that we will learn more about in a future lesson
    - Note that keras is not as ubiquitous as sklearn, so you may need to install it on your machine if you haven't used it previously



For simplicity and east of use, when we are building models in sklearn we will use the sklearn version of the data, and when building them in keras we will use the keras version

Sample Code:

```
# import and load the compressed data set from sklearn
from sklearn.datasets import load_digits
X,y = load_digits(return_X_y=True)

#plot some example images from the sklearn version of the MNIST data (in this case digits 0-4)
fig,ax = plt.subplots(1,5,figsize=(15,5))
for i in range(5):
  ax[i].imshow(X[i,:].reshape(8, 8), cmap='gray_r')
  ax[i].text(.1,.1,str(y[i]),fontsize=16)
  ax[i].set_xticks([])
  ax[i].set_yticks([])
plt.tight_layout()
plt.show()

# import the original data set version stored in keras
from keras.datasets import mnist
# now load the data
(X_train, y_train),(X_test, y_test) = mnist.load_data()

#plot some example images from the keras version of the MNIST data
fig,ax = plt.subplots(1,5,figsize=(15,8))
inds = [0,13,15,17,4] #manually chosen to get images of 5, 6, 7, 8, and 9
for i in range(5):
  ax[i].imshow(X_train[inds[i],:,:], cmap='gray_r')
  ax[i].text(2,2,str(y_train[inds[i]]),fontsize=16)
  ax[i].set_xticks([])
  ax[i].set_yticks([])
plt.tight_layout()
plt.show()
```

Notes on the above code:

- The keras version of the data set doesn't have the images in nifty numeric order like the sklearn data set, so we had to include the inds list to manually identify images associated with the sequential digits
- Note that when we load the keras version the full 60,000 training images and 10,000 test images of the data set are conveniently already labeled

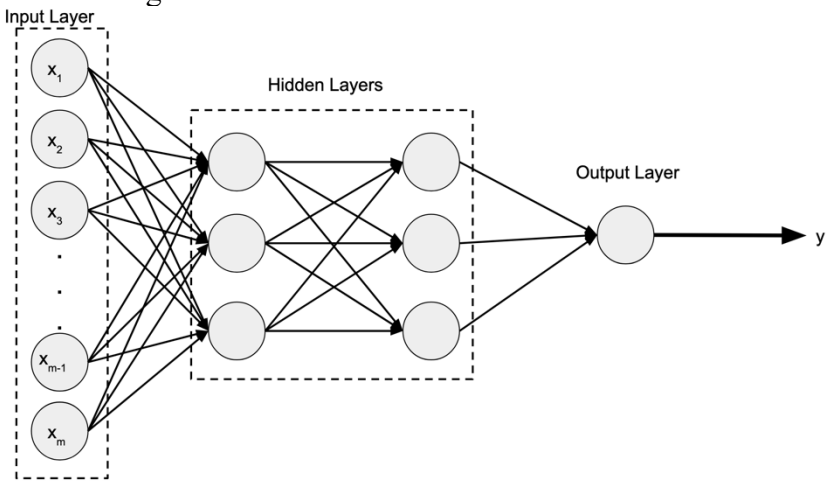
47. Multilayer Neural Networks

`Lecture Notebooks/Supervised Learning/Neural Networks/3. Multilayer Neural Networks.ipynb`

While the limitations of perceptrons to linear decision boundaries stymied neural network development for several decades, this was eventually overcome through the development of multilayer neural networks

The class of multilayer networks we examine in this lesson are known as feed forward networks, so-called because each layer feeds directly into the next one

Here's an example architecture diagram for such a neural network:



This example diagram depicts a feed forward network with 2 hidden layers, each with dimension 3, for binary classification

- We call these hidden layers because we only see what is put into the input layer and what comes out of the output layer, so in a sense what goes on in the middle layers is "hidden" to us
- Note that neural networks can have more complex architectures, but we'll get started by considering just simple feed forward networks

The output layer has a single node for binary classification and multiple nodes for multiclass classification

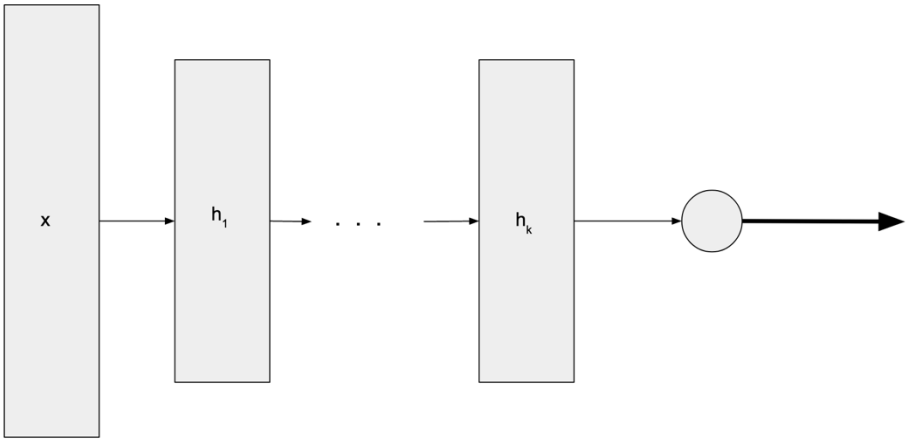
Let's now look at how to formulate the multilayer network mathematically, which will get a bit notation-heavy

- Suppose that we have n observations of m features, letting x represent a single observation as an m by 1 vector
- We'll suppose we have k hidden layers and that layer l has p_l nodes in it, taking h_l to denote a vector corresponding to hidden layer l
- Also suppose that the outer layer has o nodes
- Let W_1 be a p_1 by m weight matrix, while for $l = 2, \dots, k$ let W_l be a p_l by p_{l-1} weight matrix, and let W_{k+1} be an o by p_k weight matrix
- Finally, take Φ to be some activation function

We can then set up the recursively defined equations that are used to calculate the network output:

$$h_1 = \Phi(W_1 x) \qquad \text{Input to Hiddel Layer } l$$
$$h_{l+1} = \Phi(W_{l+1} h_l) \forall l = 1,2, \dots, k-1 \qquad \text{Hidden Layer } l \text{ to Hidden Layer } l+1$$
$$\hat{y} = \Phi(W_{k+1} h_k) \qquad \text{Hidden Layer } k \text{ to Output Layer}$$

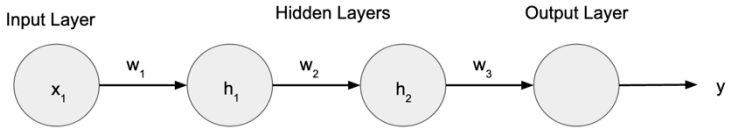
Sometimes you might see architecture diagrams that depict the nodes as large rectangles meant to represent the vectors in the recursive equations



Let's now examine the process by which we fit a multilayer network to find the optimal weight vector w

- We do so using backpropagation, which is essentially just a jargon-y way of saying the chain rule mixed with gradient descent, consists of a forward step and a backwards step

We'll demonstrate this using the simple architecture shown here:



In this architecture we have:

$$h_1 = \Phi(w_1 x_1), \qquad h_2 = \Phi(w_2 h_1), \qquad \hat{y} = \Phi(w_3 h_2)$$

The Forwards Step

- Let $w = (w_1, w_2, w_3)^T$, and as with the perceptron, initialize w with a set of random weights
- Then run a randomly selected training point, $x^{(i)}$, through the network, recording the values for each layer of the network along the way
- Thus, when the forward step is completed you have a $\hat{y}$ and $\hat{h}$s for each layer of the network

The Backwards Step

- Let our cost function (sometimes called lost function) be $C = (\hat{y} - y)^2$
- In order to update w we use gradient descent, so $w_{new} = w_{old} - \eta \nabla C(w_{old})$, where the gradient is taken with respect to w, and η is the learning rate hyperparameter
    - For the purposes of our derivation we'll assume that C is differentiable with respect to all the weights (and in practice there are workarounds for activation functions where this isn't the case)
- Using the chain rule we can find $\partial C / \partial w_1$, $\partial C / \partial w_2$, and $\partial C / \partial w_3$ to be:

$$\frac{\partial C}{\partial w_3} = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_3} = 2(\hat{y} - y)\Phi'(w_3 h_2) h_2$$

$$\frac{\partial C}{\partial w_2} = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_2} \frac{\partial h_2}{\partial w_2} = 2(\hat{y} - y)\Phi'(w_3 h_2) w_3 \Phi'(w_2 h_1) h_1$$

$$\frac{\partial C}{\partial w_1} = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_2} \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial w_1} = 2(\hat{y} - y)\Phi'(w_3 h_2) w_3 \Phi'(w_2 h_1) w_2 \Phi'\left(w_1 x_1^{(i)}\right) x_1^{(i)}$$

- We populate the values in the above expressions using what we found during the forwards step
- We then update the weights to be $w_{new} = w_{old} - \eta \nabla C(w_{old})$

We then randomly choose another training instance and repeat the cycle until we've gone through all of the training points

- Each of these cycles is called an epoch

The indexing for more complex feed forward architectures can be much more of a headache, but they all follow the basic backpropagation outline laid out above

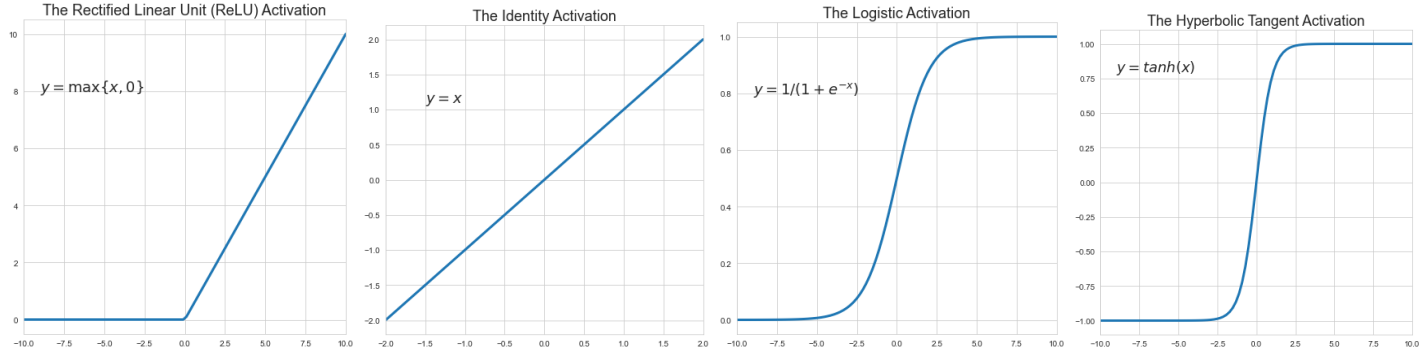
Common adjustments to the gradient descent steps:

- Sometimes in order to speed up calculations on all of the training points you'll perform batch gradient descent, in which small batches of points are run through the forwards step with the same w and then for the update you use the average of the batch's backwards step
- Instead of selecting η by hand, you can let it be a random value for each step, the idea being that it can help you get out of local minima of the cost function
    - This is known as stochastic gradient descent

Note that it is common in neural network modeling to use just a validation set, since cross-validation can take too long to be practical

In sklearn you implement multilayer network classification with MLPClassifier and multilayer network regression with MLPRegressor

- As their default activation function Φ, both of these use rectified linear unit activation ("ReLU")
- Other built-in activation functions are the identity activation ("identity"), the logistic activation ("logistic"), and the hyperbolic tangent activation ("tanh")



Identifying the network architecture that works best for you will depend, as is so often the case, on your application and the nature of your data set

- Typically you'll have to do some sort of tuning process to find the optimal architecture

It has been proven mathematically that "a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of $\mathbb{R}^n$, under mild assumptions on the activation function"

- Thus, simple neural networks can represent a wide variety of interesting functions when given appropriate parameters
    - However, the aforementioned mathematical proof does not touch upon the actual algorithmic learnability of those parameters
- Meaning that, while we can theoretically approximate any reasonable function with a high enough dimensional single hidden layer feed forward network, this is not always possible in practice

It has been found that you can trade in the height of a single hidden layer for increased depth and get similar results, and wanting to better understand the possibilities and limitations of such architecture is where the field of deep learning comes from

Deficiencies of this Method

- Feed forward neural nets can very easily overfit the training data, although this can be mitigated with a variety of techniques
    - One such technique is to add dropout layers, where some of the nodes from the previous layer are randomly selected (using a pre-set frequency rate) to be set to 0
- Gradients can vanish or explode when your networks get too deep because of the chain rule
- Convergence can be slow and difficult
- Cost functions often have many local minima in which you can get stuck when using normal gradient descent with a fixed learning rate
- May need to use powerful computers to train complicated networks (i.e., your laptop may not suffice)

Sample Code:

```
# import the model
from sklearn.neural_network import MLPClassifier

# make an mlp classifier with 1 hidden layer of 500 nodes
mlp1 = MLPClassifier(hidden_layer_sizes = (500,), max_iter=1000)
# make a second classifier with 2 hidden layers of 200 nodes each
mlp2 = MLPClassifier(hidden_layer_sizes = (200, 200,), max_iter=1000)

# fit the two classifiers
mlp1.fit(X_train_train, y_train_train)
mlp2.fit(X_train_train, y_train_train)

# import packages for comparing the two models
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix

#get training/validation accuracy & confusion matrix for the 1st model (1 hidden layer with 500 nodes)
mlp1_training_acc = accuracy_score(y_train, mlp1.predict(X_train))
mlp1_validation_acc = accuracy_score(y_val, mlp1.predict(X_val))
mlp1_confusion_matrix = pd.DataFrame(confusion_matrix(y_val, mlp1.predict(X_val)),
        columns=["predicted "+str(i) for i in range(10)],
        index=["actual "+str(i) for i in range(10)])

#get accuracies & confusion matrix for the 2nd model (2 hidden layers with 200 nodes each)
mlp2_training_acc = accuracy_score(y_train, mlp2.predict(X_train))
mlp2_validation_acc = accuracy_score(y_val, mlp2.predict(X_val))
mlp2_confusion_matrix = pd.DataFrame(confusion_matrix(y_val, mlp2.predict(X_val)),
        columns=["predicted "+str(i) for i in range(10)],
        index=["actual "+str(i) for i in range(10)])
```

Notes on the above code:

- Since we didn't specify the activation function here, both of the models are using the default "relu"
- When specifying layer sizes, note that you need to include a comma after the last number

48. keras

Lecture Notebooks/Supervised Learning/Neural Networks/4. keras.ipynb

As mentioned in an earlier lesson, keras is not as ubiquitous as sklearn, so you may need to install it on your machine if you haven't used it previously

From the documentation: "Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result as fast as possible is key to doing good research."

Here we'll go through an extended worked example illustrating how to build an MNIST classifier in keras

- Since this is less of a conceptual lesson and more of a practical coding exercise, I'm mostly just going to copy over the notebook here
- When reviewing this lesson in practice it'd probably be best to just look at the notebook

We will mimic our sklearn networks from the last lesson and build an MNIST classifier.

```
# import  MNIST dataset stored in keras
from keras.datasets import mnist
# load the data
(X_train, y_train),(X_test, y_test) = mnist.load_data()
print("Original shape of X_train", np.shape(X_train))
```

```
Original shape of X_train (60000, 28, 28)
```

We'll now reshape the data so that it goes from a 28 by 28 grid of pixels to a single column of length 28*28 and scale it such that the maximum value becomes 1 instead of 255

```
X_train = X_train.reshape(-1,28*28)
X_test = X_test.reshape(-1,28*28)
print("The new shape of X_train is", np.shape(X_train))
print("The new shape of X_test is", np.shape(X_test))
X_train = X_train/255
X_test = X_test/255
```

```
The new shape of X_train is (60000, 784)
The new shape of X_test is (10000, 784)
```

Before we can build our keras model, we first need to import the necessary model components

```
from keras import models
from keras import layers
from keras import optimizers
from keras import losses
from keras import metrics
from keras.utils.np_utils import to_categorical
```

What we called feed forward networks in the last lesson are also called dense networks (because they are fully connected graphs)

- We'll now walk through the process of making a dense neural networks using keras

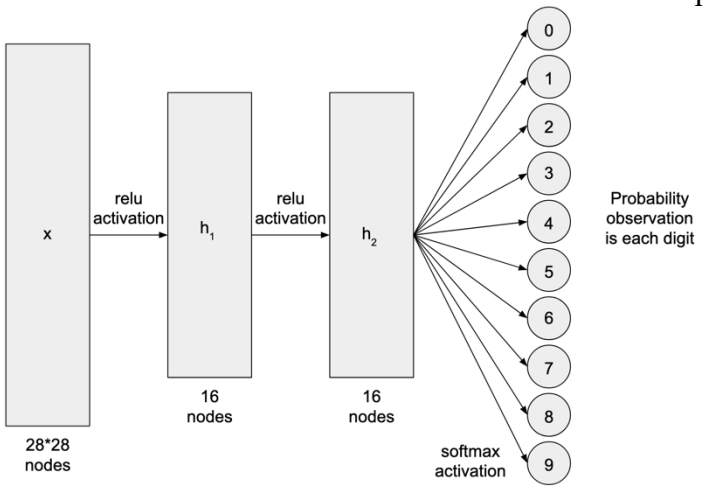
Step 1: Make an empty model

- Note that "sequential" here means we'll be making a group of a linear stack of layers

```
# we first make an empty model
model = models.Sequential()
```

Step 2: Add layers to the model

- We will be building the neural network architecture shown below in this example



- If you're using a jupyter notebook, it's very important to only run the layer-adding code chunk once, since it is going to add the prescribed layer to your model every time you run it
- You add layers to the model with .add()
  - Note that you need to specify the input shape only for the first layer
  - As with sklearn, syntax dictates you include a comma after the input shape
- We use 'softmax' activation for the output layer so that we can get probability estimates

```
##### ONLY RUN THIS ONCE! #####
# add the first layer, using 16 nodes and relu activation, and passing along the input shape information
model.add(layers.Dense(16, activation='relu', input_shape=(28*28,)))
# now add a second layer with relu activation and 16 nodes
model.add(layers.Dense(16, activation='relu'))
# finally, add the 10-node (corresponding to digits 0-9)  output layer with softmax activation for probabilities
model.add(layers.Dense(10, activation='softmax'))
```

Step 3: Compile the model with an optimizer, loss, and metric

- The optimizer we'll use is "rmsprop", an algorithm implemented by keras to perform the backgpropagation step in neural network fitting
- The loss we'll use is "categorical entropy", which stems from information theory and is a common/popular choice for classification problems
- The metric we'll use is simply accuracy, although you have the option to include multiple metrics
- Note that the keras documentation lists other built-in options for all three parameters, and you can also define your own custom inputs

```
# now let's compile the network
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics =['accuracy'])
```

Step 4: Fit the model on the training data

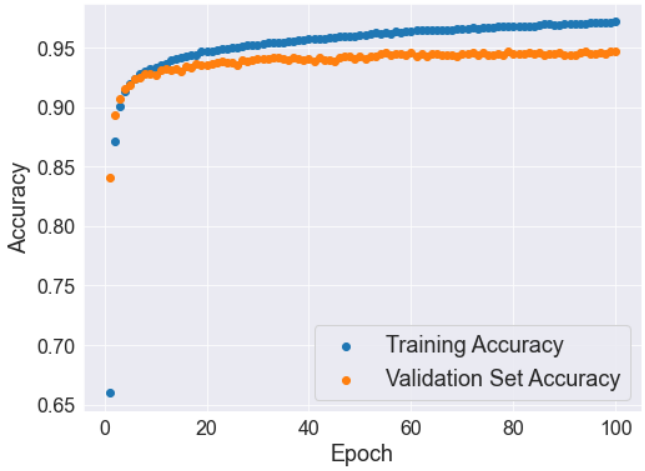
- To fit the model we'll look at 100 epochs using batch gradient descent with 512 observations per batch
- First we'll need to create our validation set, and when we pass it to our model we'll need to convert it to categorical outputs using the keras function to_categorical
    - This function basically just one-hot encodes the data
- Note that keras will print out a series of progress bars while it runs through the fitting process

```
# make the validation set
from sklearn.model_selection import train_test_split
X_train_train,X_val,y_train_train,y_val = train_test_split(X_train, y_train, test_size=0.2, shuffle=True,
                                                           stratify=y_train, random_state=440)

# now fit the model and store the training history
history = model.fit( X_train_train, to_categorical(y_train_train), epochs = 100,
                    batch_size = 512, validation_data=(X_val,to_categorical(y_val)) )
```

Step 5: Examine epoch history loss and acccuracy

- The data stored in history includes a dictionary with the training and validation losses/accuracies from which we can easily plot the metrics as a function of epoch
- Looking at these metrics can allow us to choose a training period (i.e., number of epochs) for a neural network as well as compare performance between two different networks
    - In this example it seems that we start to overfit the data somewhere between epochs 20 and 30
    - We identify this by noting where the training and validation metrics begin to diverge

```
history_dict = history.history
print(history_dict.keys())
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

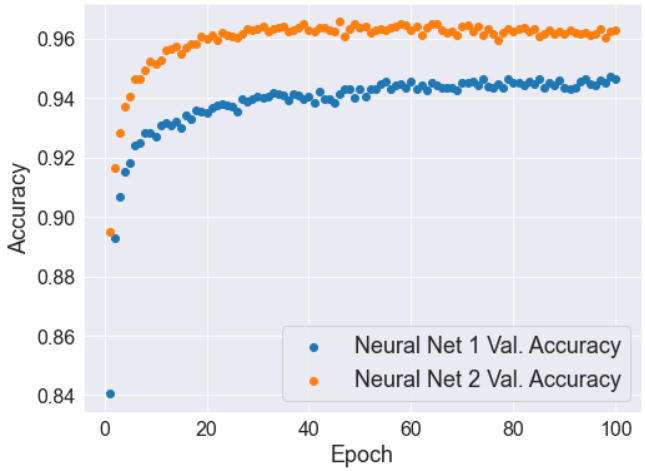


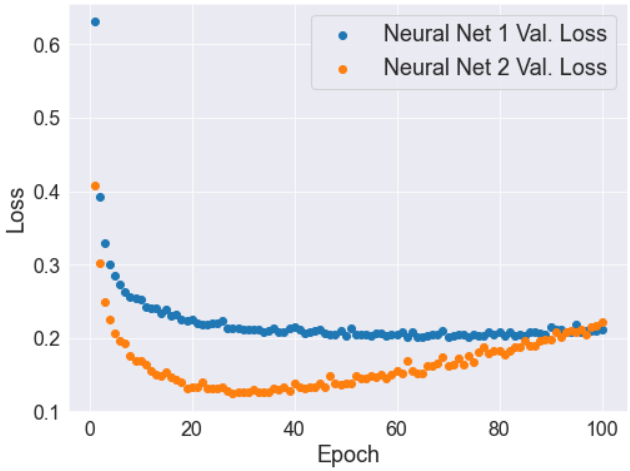
Step 6: Tune the model architecture

- Let's build a second network using two 32-node hidden layers and compare it with our original network that has two 16-node hidden layers
    - We'll keep the rest of the model parameters the same

```
# initialize the second model
model2 = models.Sequential()
```

```
# change the structure of the first model so that we now have two 32-node hidden layers
model2.add(layers.Dense(32, activation='relu', input_shape=(28*28,)))
model2.add(layers.Dense(32, activation='relu'))
model2.add(layers.Dense(10, activation='softmax'))

# compile the model
model2.compile(optimizer = 'rmsprop', = 'categorical_crossentropy', metrics = ['accuracy'])

# train it and store metrics for comparison
history2 = model2.fit(X_train_train, to_categorical(y_train_train), epochs = 100,
            batch_size = 512, validation_data = (X_val, to_categorical(y_val)))

history_dict2 = history2.history
```





Step 7: Selecting an architecture

- From the above plots, it looks like the 32x32 network outperforms the 16x16 network, so let's find the epoch for the 32x32 network that resulted in the lowest validation loss and use that for our final model

```
print("The epoch that had the lowest model 2 validation loss was",
   range(1,102)[np.argmin(history_dict2['val_loss'])])

# make/train net 2 using this optimal epoch
model2 = models.Sequential()
model2.add(layers.Dense(32, activation='relu', input_shape=(28*28,)))
model2.add(layers.Dense(32, activation='relu'))
model2.add(layers.Dense(10, activation='softmax'))
model2.compile(optimizer = 'rmsprop', loss = 'categorical_crossentropy', metrics = ['accuracy'])
history2 = model2.fit(X_train,
            to_categorical(y_train),
            epochs = range(101)[np.argmin(history_dict2['val_loss'])],
            batch_size = 512,
            validation_data = (X_val, to_categorical(y_val)))
history_dict2 = history2.history
The epoch that had the lowest model 2 validation loss was 28
```

Step 8: Predicting on the test set

- For the purposes of this lesson, we'll assume we are now done with the model selection process, so we can now use our model to make predictions in a fashion nearly identical to that of sklearn
    - Note that model.predict(X_test) produces a set of probabilities for each test observation (i.e., the probability that it is a specific digit 0-9)
    - For our prediction we'll generally just want to take the one with the highest probability

```
# this gives the probability for each image in the test set being digit 0-9, so each point has an associated array
print(model2.predict(X_test))
[[1.1564995e-06 1.7546260e-11 1.3643604e-06 ... 9.9780113e-01
  3.6319830e-07 9.8308274e-06]
 [4.8516970e-08 1.7183842e-03 9.9827754e-01 ... 1.7715241e-12
  1.6297822e-06 2.8923276e-15]
 [2.2510280e-06 9.9844968e-01 1.2244676e-04 ... 9.8002213e-04
  2.5528760e-04 5.7948323e-06]
 ...
 [5.8266050e-09 4.8411105e-11 1.2913644e-10 ... 3.2273707e-05
  1.5033721e-05 3.7391433e-03]
 [2.1717947e-09 1.1105325e-07 1.8425939e-10 ... 1.4940962e-08
  5.7690368e-06 6.7102626e-09]
 [4.0267051e-10 5.4579181e-14 5.2682875e-10 ... 6.3181224e-14
  1.4230515e-08 4.2205134e-14]]

# this simply returns the most likely classification for each test point, which can be used for calculating accuracy
preds = np.argmax(model2.predict(X_test), axis=1)
print(preds)
[7 2 1 ... 4 5 6]

from sklearn.metrics import accuracy_score
print(np.round(100*accuracy_score(y_test, preds),2))
96.51
```

Looks like we're getting over 96% accuracy, not bad!

49. Introduction to Convolutional Neural Networks

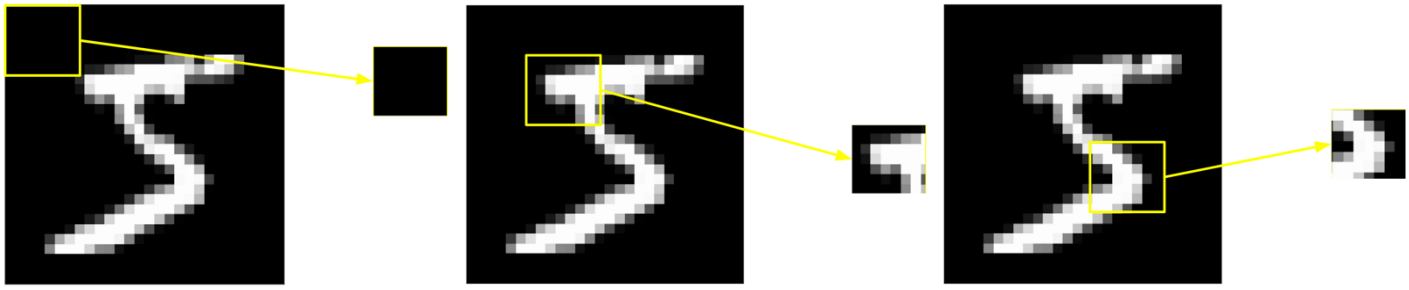
`Lecture Notebooks/Supervised Learning/Neural Networks/5. Introduction to Convolutional Neural Networks.ipynb`

Convolutional neural networks were developed to work with grid-based data sets, and are therefore particularly useful for classification problems involving images and their natural pixel-based grid structure

- The basic idea underlying a convolution neural network is "What if we paid attention to small portions of an image instead of looking at the entire thing at once?"

While a feed forward model works by taking a weighted sum of the values in every single pixel (meaning it looks at the entire image all at once), a convolutional neural network slides a small square grid along the image, only focusing on what is enclosed in the small grid at each step in the sliding process

- As we'll see shortly, this sliding is actually a series of weighted sums whose results themselves get stored in a grid



Let's look at the three parts of a convolutional neural network

- Rather than go through a wild adventure in notation-tracking, we'll use some arbitrary example grids

Part 1 – The Convolutional Layers

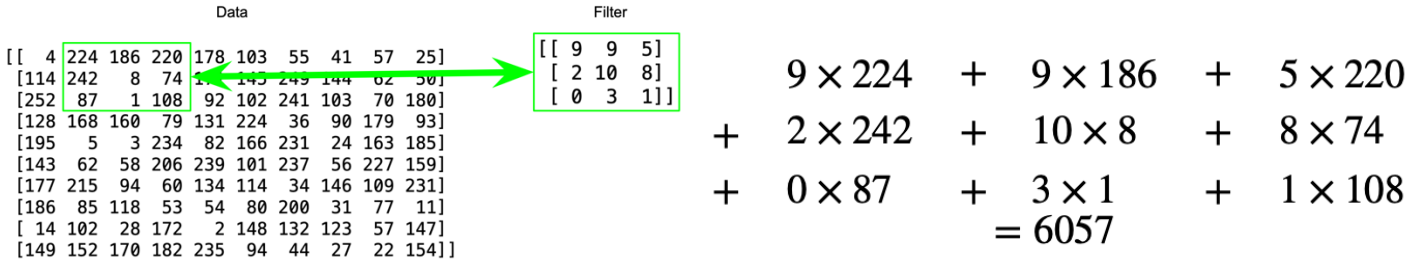
Suppose that our image is a 10x10 grid represented by the 2D array below

```
[[  4 224 186 220 178 103  55  41  57  25]
 [114 242   8  74 171 145 249 144  62  50]
 [252  87   1 108  92 102 241 103  70 180]
 [128 168 160  79 131 224  36  90 179  93]
 [195   5   3 234  82 166 231  24 163 185]
 [143  62  58 206 239 101 237  56 227 159]
 [177 215  94  60 134 114  34 146 109 231]
 [186  85 118  53  54  80 200  31  77  11]
 [ 14 102  28 172   2 148 132 123  57 147]
 [149 152 170 182 235  94  44  27  22 154]]
```

We'll slide a 3x3 grid (called a filter) around this array, in this example using some randomly chosen weights to demonstrate the basics of the method

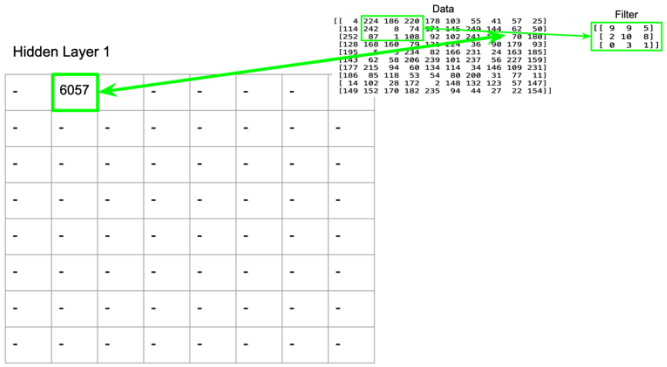
- This is a fairly standard size for the filter, although 5x5 is also common
- There are 8x8 possible grid locations for a 3x3 grid over a 10x10 grid
- More generally, if your grid has dimensions $L \times B$ and your filter is a $F \times F$ grid, then there will be $(L - F + 1) \cdot (B - F + 1)$ potential positions

With this filter over the green highlighted points, we "focus" on this square of the image using a dot product between the two arrays



The outputs of these 8*8 dot products are then fed into an activation function and stored in their own 8x8 grid, which makes up the first hidden layer of the network

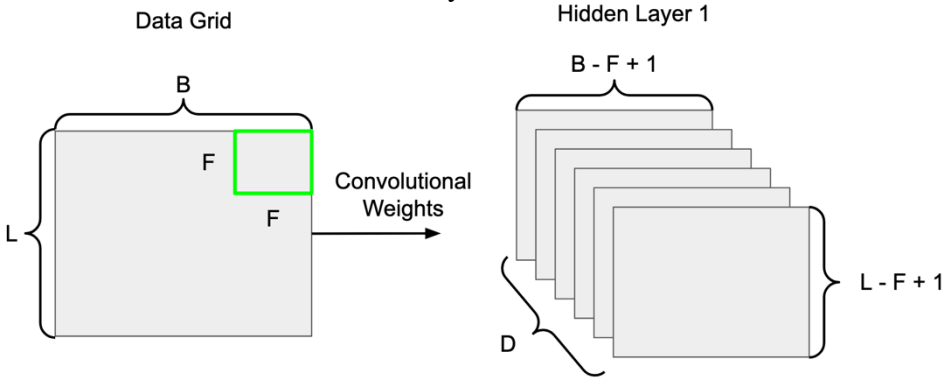
- So in this simple example where we're using the identity function as our activation function, we'd have 6057 as the (1, 2) entry of the hidden layer



In a real convolutional neural net our hidden layer would have 3 dimensions, the first two representing the $(L - F + 1) \cdot (B - F + 1)$ grid, and the third indicating that we do this sliding grid process multiple times

- The output from each sliding grid process is stored in the depth of the network
- If you did the filter process 16 times, for example, the dimensions of the hidden layer would be $(L - F + 1) \cdot (B - F + 1) \cdot 16$
- An activation is then applied to each value in the hidden layer, with the ReLU function being the usual choice

Summarizing the structure of the first convolutional layer:

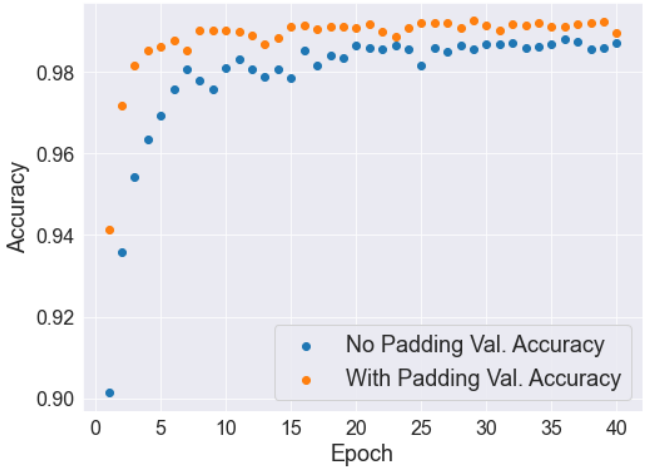


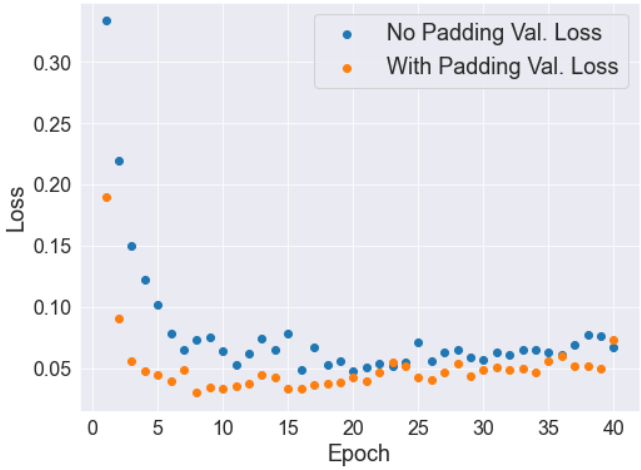
Padding and Stride Values

- You may notice that the sliding grid process pays less attention to the borders of your image than the inside of the image
  - To prevent loss around the borders it is common to add padding cells around the outside of your input grids
  - These are extra rows and columns consisting of all 0s

- You can also choose different stride values for the sliding grid
    - The convolutional layers stride value is how many grid points you slide to the right and down
    - A stride value of 1, what we used in the above example, is pretty typical, but in practice you can choose any strid value you'd like

Including padding can provide significant improvement for a convolutional neural network, for example in the keras-implemented CNN described in the sample code below:





Part 2 – The Pooling Layers

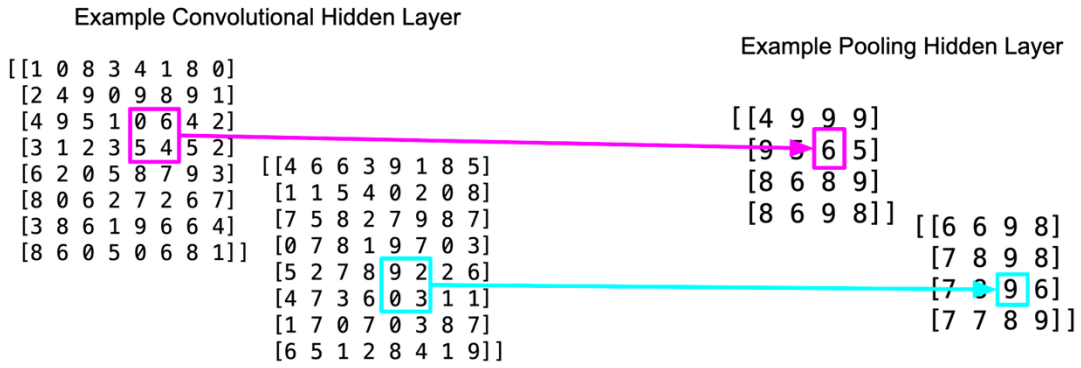
It has become common practice to add what are known as pooling layers after your convolutional layers

- These layers shrink the size of our grids in order to down-sample our observations and shrink the number of parameters needed to fit the model

Pooling layers work by sliding a square grid over each grid in the convolutional layer, keeping only the maximum value captured in the square

- The most common pooling square size is 2x2 with a stride value of 2, although you could change this if you feel so inclined

Looking at an example 8x8x2 convolution layer using 2x2 pooling squares, we can see this in practice:



You could conceivably choose alternative pooling operations like taking the average of the entries in the square, but max pooling has become more or less standard convention since it has been found to work better in practice

- Note that pooling hidden layers have the same depth as their input layer, in contrast to the convolutional layers which generally have a larger depth (i.e., more dimensions) than their input layer

Part 3 – The Fully Connected Layer

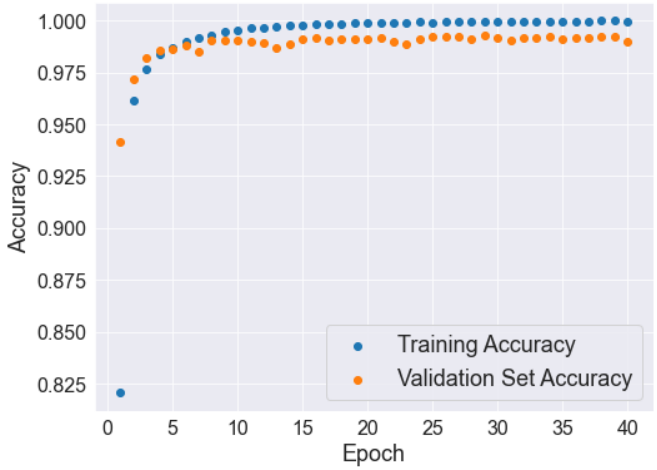
After building a number of alternating convolutional and pooling layers, you end the network with a fully connected layer, the kind of layer we have dealt with in prior lessons

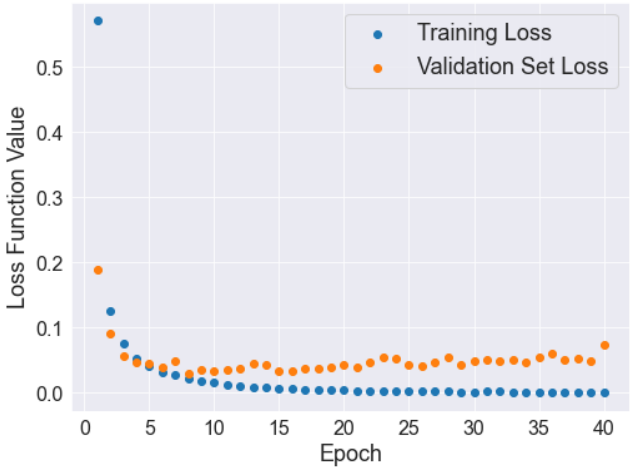
- We can therefore think of the convolutional and pooling layers as a sort of preprocessing step for the dense neural network we build as the last step

Prior to going to the fully connected layer, we will flatten (make into a single column vector) the hidden layer

- Note this is one of the main reason we pool in earlier layers
- Without pooling, the number of parameters we'd need to fit in the fully connected layers would be huge

In the sample code below we'll illustrate how to implement a padding-inclusive convolutional neural network in keras that yields the following training metrics as a function of epoch:





Sample Code:

```
# import necessary keras components
from keras import models
from keras import layers
from keras import optimizers
from keras import losses
from keras import metrics
from keras.utils.np_utils import to_categorical

# need to reshape data for use with convolutional neural net (different from dense neural nets)
X_train = X_train.reshape((60000,28,28,1))
X_test = X_test.reshape((10000,28,28,1))

# now that we've got the right shape, make validation set
from sklearn.model_selection import train_test_split
X_train_train,X_val,y_train_train,y_val = train_test_split(X_train, y_train, test_size=0.2,
                                    shuffle=True, stratify=y_train, random_state=440)

# make an empty base model
model = models.Sequential()

# add our first convolutional layer
model.add( layers.Conv2D(32, (3,3), activation='relu', input_shape = (28,28,1), padding='same' ))
```

```
# add our first max pooling layer
model.add( layers.MaxPooling2D( (2,2), strides=2) )

# add a few more layers, alternating between conv. and pooling layers, this time using 64-node conv. layers
model.add( layers.Conv2D(64, (3,3), activation='relu', input_shape = (28,28,1)) )
model.add( layers.MaxPooling2D( (2,2), strides=2) )
model.add( layers.Conv2D(64, (3,3), activation='relu', input_shape = (28,28,1)) )
model.add( layers.MaxPooling2D( (2,2), strides=2) )

# now add the fully connected layer
# need to flatten the data
model.add(layers.Flatten())
# add a single dense hidden layer, here using 64 nodes
model.add(layers.Dense(64, activation='relu'))
# finally include an output layer, here with 10 nodes to match the digits 0-9
model.add(layers.Dense(10, activation='softmax'))

# examine the model architecture
model.summary()

# compile the network
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
# fit the model (this will take a little while, so we'll limit ourselves to 40 epochs)
epochs=40
history = model.fit(X_train_train, to_categorical(y_train_train), epochs=epochs,
          batch_size=512, validation_data=(X_val,to_categorical(y_val)))
history_dict = history.history
```

Notes on the above code:

- This assumes you've previously imported the MNIST data set using keras and done the standard preprocessing sequence (reshaping, scaling, validation split, etc.)
- When specifying the convolutional layer Conv2D(32) indicates that we want a convolutional layer with depth 32, (3,3) is our sliding grid size, we're using 'relu' activation, and since we're working with 28x28 grayscale images our input shape is (28, 28, 1)
    - If we instead had RGB images we'd specify an input shape of (28, 28, 3)
    - Including padding='same' tells keras to include border padding for the images, which tends to help performance since border regions are otherwise neglected by the model
- Specifying the max pooling layer with (2, 2) and strides=2 tells keras that we are using a 2x2 pooling squares with a stride value of 2, the common choice illustrated above
- Looking at the training/validation accuracy/loss plots above we'd probably use 10 or so epochs in our final model, at which point we could obtain predictions using the same model.predict(X_test) syntax we've seen previously

50. Future Directions with Neural Networks

`Lecture Notebooks/Supervised Learning/Neural Networks/6. Future Directions.ipynb`

What we've learned about so far:

- Perceptrons
- Feed forward networks
- Basic convolutional neural networks

Not surprisingly, this barely scratches the surface of the field of neural networks

Some example topics we haven't been able to touch on:

Theoretical

- Recurrent neural networks
- Autoencoders
- Transformers
- Generative adversarial networks
- And more!

Practical

- More applications of keras
  - Documentation: https://keras.io/
- tensorflow
  - Documentation: https://www.tensorflow.org/api_docs
- PyTorch
  - Documentation: https://pytorch.org/

Potentially useful resources for future learning:

Theoretical

- Neural Networks and Deep Learning

Practical

- Deep Learning with Python is good for learning how to implement things in keras
- Hands-On Machine Learning with Scikit-Learn, Keras and Tensorflow is a good general purpose python machine learning book
- Deep Learning with PyTorch is a good resource for PyTorch

51. *t*-distributed stochastic neighbor embedding (tSNE)

`Lecture Notebooks/Unsupervised Learning/Dimensionality Reduction/4. tSNE.ipynb`

t-distributed stochastic neighbor embedding (or tSNE) reduces the dimension of a set of *m* features, *X*, typically down to 2 or 3 dimensions for the purposes of data visualization

- A primary goal of tSNE is to ensure that points close to one another in the higher dimensional space are also close to one another in the lower dimensional projection
- The way it does so is to estimate pairs of probability distributions in such a way as to ensure that they are as close to one another as possible

The basic outline of the algorithm is as follows:

1. We convert the Euclidean distance for all points x_i, x_j into a conditional probability $p_{i,j}$
    - This is done by imagining a Gaussian distribution around x_i and then comparing the "normal distance" of x_j vs the sum of all other "normal distances"
    - The precise mathematical formula for this is thus

$$p_{j|i} = \frac{\exp\left(-\left\|-x_i - x_j\right\|^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp(-\|-x_i - x_k\|^2 / 2\sigma_i^2)}$$

    - Think of $p_{j|i}$ as the probability that x_i would choose x_j as its neighbor, and we take $p_{i|i} = 0$

2. For every point x_i in high dimensional space we will have a low dimensional counterpart y_i onto which we map x_i
    - Similar to $p_{j|i}$ we have $q_{i|j}$, which gives the probability that y_i would choose y_j as its neighbor and for which we also have $q_{i|i} = 0$
    - This has the mathematical form

$$q_{j|i} = \frac{\left(1 + \left\|y_i - y_j\right\|^2\right)^{-1}}{\sum_{k \neq i}(1 + \|y_i - y_k\|^2)^{-1}}$$

    - This is where the "*t*-distributed" part of tSNE's name comes from, since the numerator and denominator in the above expression come from the probability density for the *t*-distribution with 1 degree of freedom

3. If we're preserving these pairwise distances well, then $p_{j|i}$ should be close to $q_{j|i}$, so we now minimize a cost function that measures the difference between $p_{j|i}$ and $q_{j|i}$ using gradient descent
    - The specific cost function is the Kullback–Leibler divergence
    - Once completed, the optimal y_i values are spit out by the algorithm

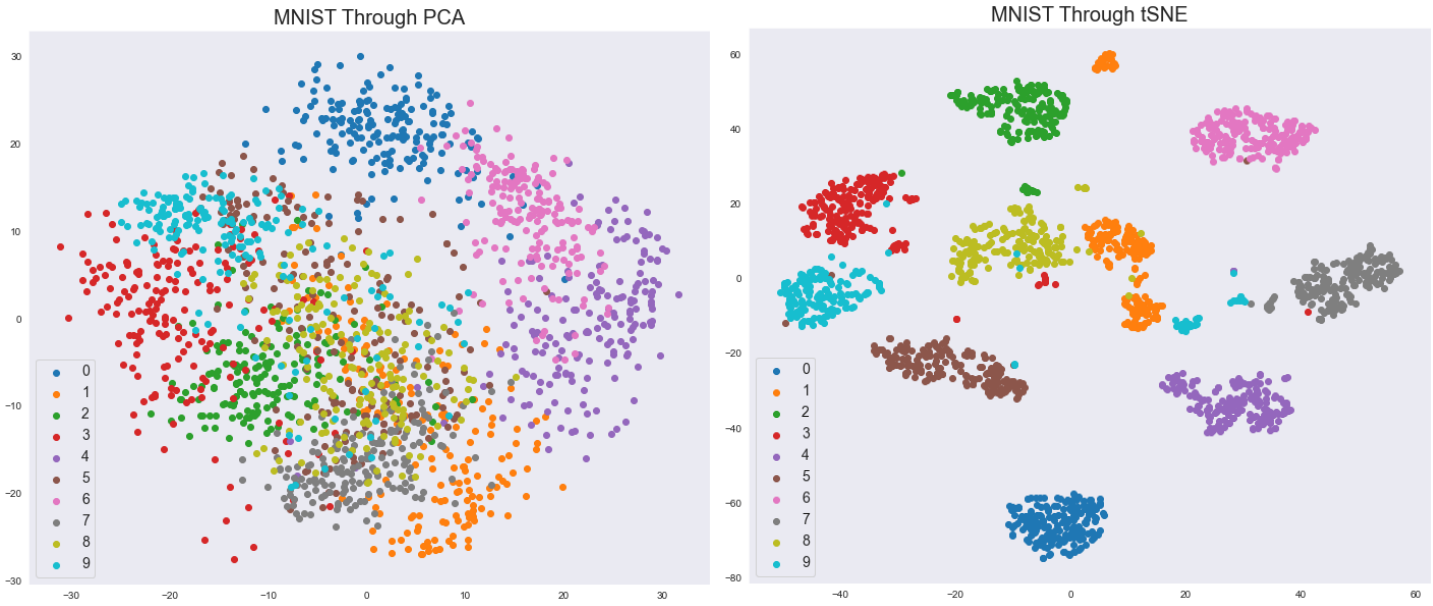
The σ_i values in Step 1 are determined based on a hyperparameter called the perplexity of the tSNE

- With smaller perplexity values the algorithm tends to focus on more local behavior, while larger perplexities lead to a greater focus on more global behavior
- Perplexity values between 5 and 50 tend to work well

We implement tSNE in sklearn using the TSNE package

There are major drawbacks that limit its utility elsewhere, but when used purely for data visualization tSNE can offer a significant improvement over PCA

- Looking for example at the MNIST data:



Disadvantages of tSNE

- The "S" stands for "Stochastic", meaning that your results change slightly each time
  - You could get around this to an extent by setting a random state
- You can't really use this to make predictions on new data
  - Unlike PCA there isn't a procedure that will map new points onto the lower dimensionality space
- The magnitude of the distances between clusters shouldn't be interpreted as a meaningful metric
  - Basically local distances are preserved well, but things that are far away get distorted in the projection process
- tSNE results should not be used as statistical evidence or proof of something since it is not a formal statistical test
- Sometimes tSNE can produce clusters on data that are not actually clustered in the original data space
  - Thus, it is good to run the data through tSNE a few different times with different perplexities to ensure that the clustering persists

Sample Code:

```
# import TSNE
from sklearn.manifold import TSNE

# make the tSNE object
tsne = TSNE(n_components=2)
# get the transformed data
X_tsne = tsne.fit_transform(X)
```

Notes on the above code:

- The n_components=2 parameter is telling TSNE to project down to 2-D space
  - This is the default value, but there might be applications where you'd rather use 3 dimensions
- In this example we have used the default perplexity value (30 for this package), but if we wanted to instead use a perplexity value of, say, 20 we would call TSNE(2, perplexity=20)

52. What is Clustering?

`Lecture Notebooks/Unsupervised Learning/Clustering/1. What is Clustering.ipynb`

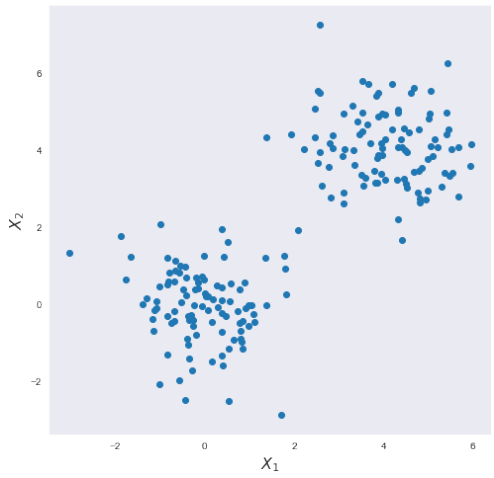
Another common unsupervised learning task is clustering

In clustering we look to identify groupings of similar points in otherwise unlabeled data, X

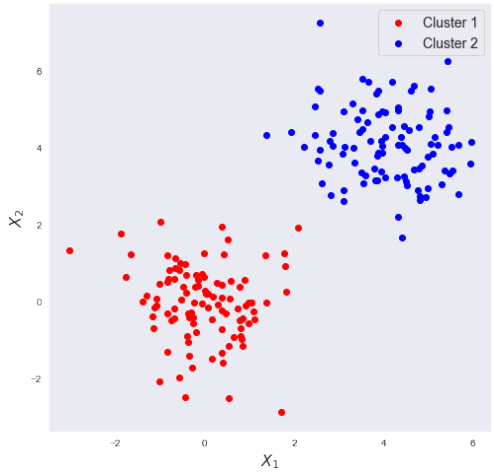
This could be done for any number of reasons, including:

- Identifying market segments
- Locating similar product users
- etc.

As a simple toy example, consider the following data X:



Given this data, a clustering algorithm might produce something like the following:



This is, of course, a painfully simple example

- In the next lessons we'll look at some more complex algorithms with real-world utility

53. *k*-Means Clustering

`Lecture Notebooks/Unsupervised Learning/Clustering/2. k Means Clustering.ipynb`

Note that this method has basically nothing to do with the *k*-nearest neighbors algorithm discussed earlier

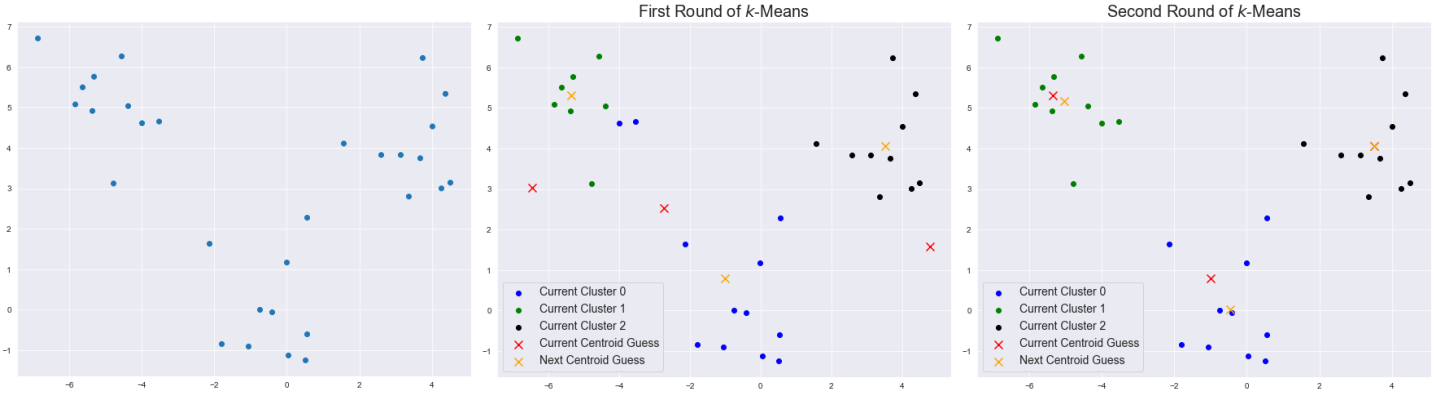
Suppose we have n observations of m features, X, that we suspect could reasonably be segmented into k groups

- The *k*-means algorithm provides us a mechanism for attempting to find these groups

In *k*-means clustering we use the following procedure

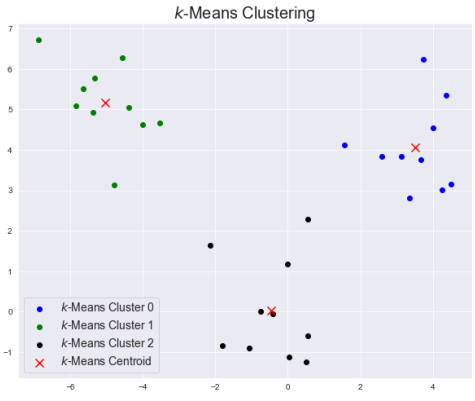
1. Select k initial points as a first guess for the k different centroids for each group, defined to be the average position of all the points within a group
    - Hence the "mean" in *k*-means clustering
2. Group all n points according to which centroid is closest
3. Recalculate the k centroids, using the groups found in Step 2
4. Repeat Steps 2 and 3 until you reach a stage where no observations change groups

Let's look at a practical example where we use *k*-means clustering to group some toy data:



For this example we'd be finished after just the two rounds shown above, since the three groups are now effectively separated and additional rounds would not cause points to change groups

We implement *k*-means clustering in sklearn with KMeans, and using this functionality we easily recover the same groupings found above:



For this example we knew to choose $k = 3$ beccause we'd randomly generated the points around three locations

- For real-world data, though, we will typically need to choose k
- Typically you will have to run the algorithm multiple times for different values of k and examine some metrics to determine which value of k is the "best"
    - Here we'll look at two potential metrics for making this choice

The first approach we can take to determine an appropriate k value is called the elbow method

- In this method we calculate the inertia of the resulting clustering for each value of k and then look for an elbow in the plot of inertia against k

For a given clustering with k clusters, the inertia is defined as

$$\sum_{i=1}^{n} \text{dist}\left(X^{(i)}, c^{(i)}\right)^2$$

where $X^{(i)}$ is the i^{th} observation in the data set, $c^{(i)}$ is the centroid of the cluster to which observation i is assigned, and $\text{dist}(a, b)$ denotes the distance between points a and b (typically the Euclidean distance)

We think of clustering with low inertia as being good, with the caveat that we cannot simply choose the value of k that gives the lowest inertia

- I.e., by setting $k = n$ we could arbitrarily get an inertia of 0, but that would not be useful

In practice we get inertia values from sklearn's KMeans object using .inertia_

The second approach we'll look at is what's known as the silhouette method

The silhouette score for a given observation I is defined to be

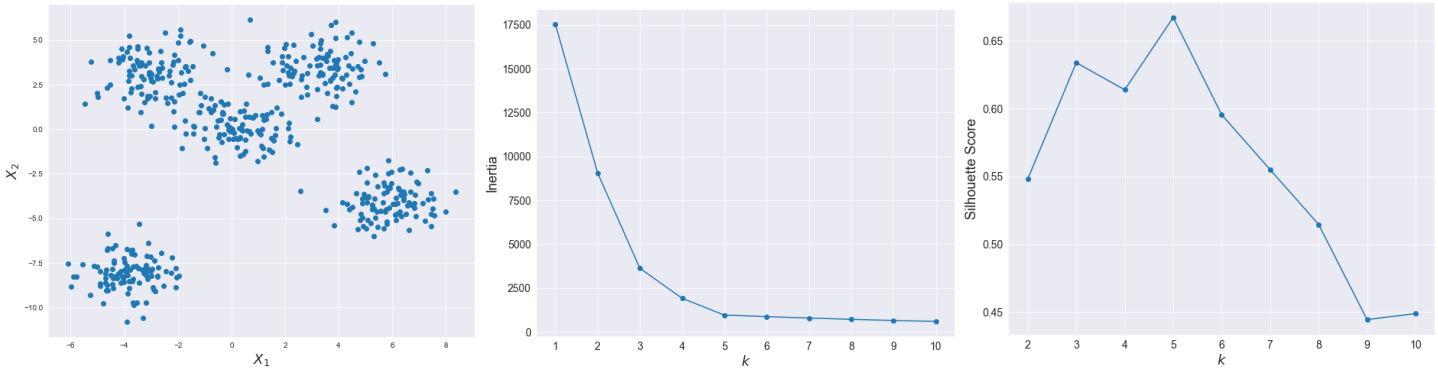
$$\frac{b - a}{\max(a, b)}$$

where a is the average distance between observation i and the rest of the points in its assigned cluster, and b is the average distance between observation i and the points in the next closest cluster

A higher silhouette score is indicative of a "good" clustering, and we will generally want to compare the average silhouette score over all n observations for various values of k

In sklearn the silhouette_score function gives the averaged cluster score, and silhouette_samples gives the scores for each individual observation

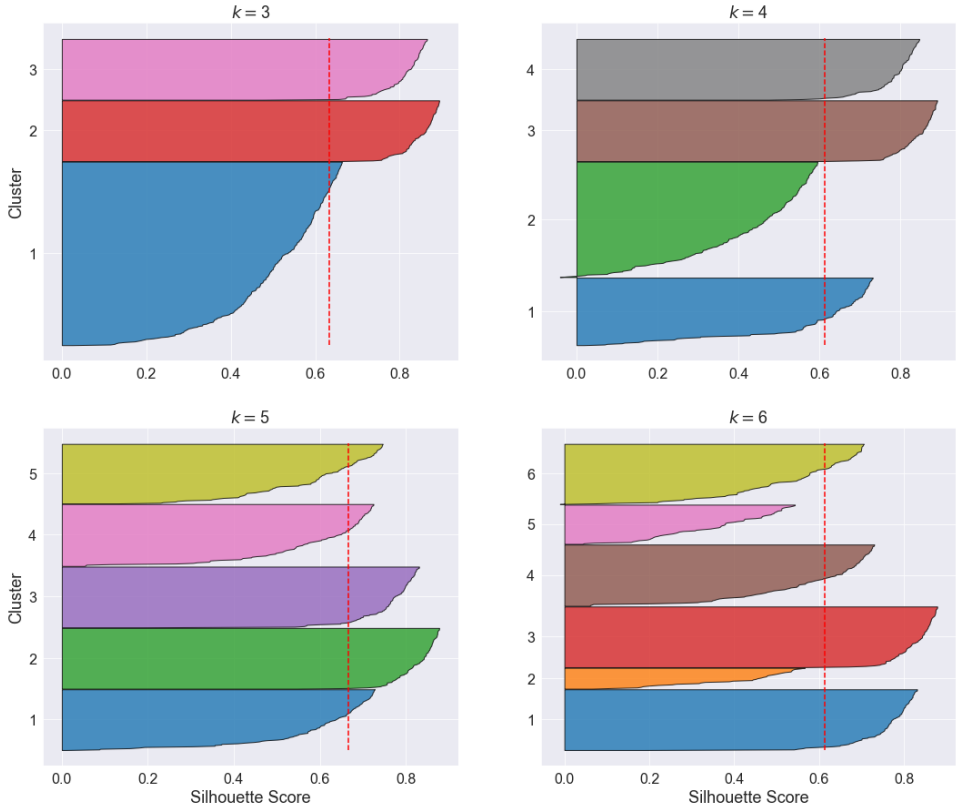
Here is what the inertia and silhouette score look like for clusters of varying k values fit on the sample data shown in the leftmost plot below:



It can also be helpful when selecting k to look at what is known as the silhouette diagram, which plots the distribution of silhouette scores for each cluster along with the sample average score

- Our goal with such plots is to avoid "bad" clusters, or those with distributions that lie completely below the average score line like the 2nd and 5th $k = 6$ clusters shown below

    - Similarly, clusters like the 1st $k = 3$ cluster where the vast majority of points lie below the average score line are also undesirable
    - Points with negative silhouette scores, like the 2nd $k = 4$ cluster shown below are particularly strong negative indicators
- See associated notebook for the code needed to produce the silhouette diagram shown below (for the same sample data as above)



Thus, in this example we would choose $k = 5$ because:

- It is located near the elbow in the inertia plot
- It has the highest silhouette score
- It has no "bad" (or nearly bad) clusters in the silhouette diagram

k-means has been shown to be a fast and scalable clustering algorithm that's easy to implement, however it does have some limitations:

- Random initialization of centroids can lead to suboptimal solutions
    - A common approach to deal with this is to run the algorithm multiple times with a different random initial set of centroids each time, and then choose the run that provided the lowest inertia
    - In sklearn the default is to run the *k*-means algorithm 10 times
        - This can be changed with the n_init argument
- Choosing *k* can be tricky
    - The elbow method is a bit course
    - The silhouette method can be computationally expensive
- Can be ill-behaved when:
    - Clusters have varying sizes
    - Clusters have different densities or nonspherical shapes
    - Features are on different scales
        - Always scale your data prior to fitting KMeans to combat this

Sample Code:

```
# import KMeans
from sklearn.cluster import KMeans

# make a KMeans object using k=3
kmeans = KMeans(3)

# fit the kmeans object
kmeans.fit(X)

# get the clusters
clusters = kmeans.predict(X)

# now plot the final clustering
plt.scatter(X[clusters==0,0], X[clusters==0,1], c='b', label="$k$-Means Cluster 0")
plt.scatter(X[clusters==1,0], X[clusters==1,1], c='green', label="$k$-Means Cluster 1")
plt.scatter(X[clusters==2,0], X[clusters==2,1], c='k', label="$k$-Means Cluster 2")
# include the centroid locations
plt.scatter(kmeans.cluster_centers_[:,0],
        kmeans.cluster_centers_[:,1],
        c='r', marker='x', s=100, label="$k$-Means Centroid")
plt.legend(fontsize=14)
plt.title("$k$-Means Clustering", fontsize=20)
plt.show()

# look at the cluster's inertia
print( kmeans.inertia_ )

# show the cluster wide silhouette score
print( silhouette_score(X, clusters) )
# show scores for each observation in X
print( silhouette_samples(X, clusters) )
```

Notes on the above code:

- We get the centroids for the final clusters using cluster_centers

54. Hierarchical Clustering

`Lecture Notebooks/Unsupervised Learning/Clustering/3. Hierarchical Clustering.ipynb`

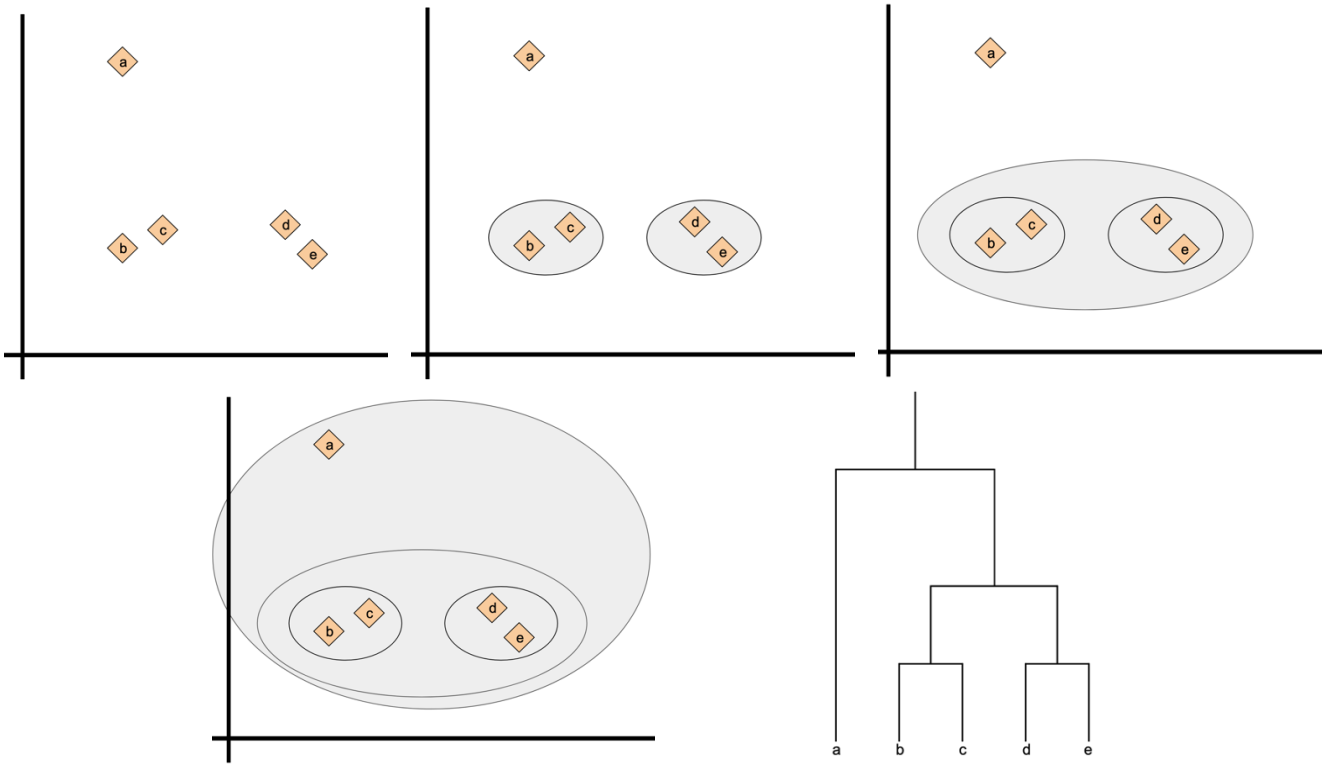
Hierarchical clustering is another technique for grouping n observations of m features stored in a matrix X

- A nice feature of hierarchical clustering, in contrast to k-means, is that we do not need to guess a number of clusters prior to fitting the algorithm
- Instead we use a dendrogram (explained below) to make an informed choice after running the algorithm

In hierarchical clustering you start from each observation being its own cluster and slowly work your way to having every observation in a single cluster

- This is done by combining clusters according to an inter-cluster distance measure that you determine prior to fitting the algorithm that we'll call d
- Starting from $d = 0$ you slowly increase d and when any pair of clusters are a distance d apart from one another, you combine them into a larger cluster
- Continue increasing the value of d until you have a final cluster that encompasses all points

Let's demonstrate this idea with a series of sketches:



The clustering information is stored in a dendrogram (the lower right chart shown above)

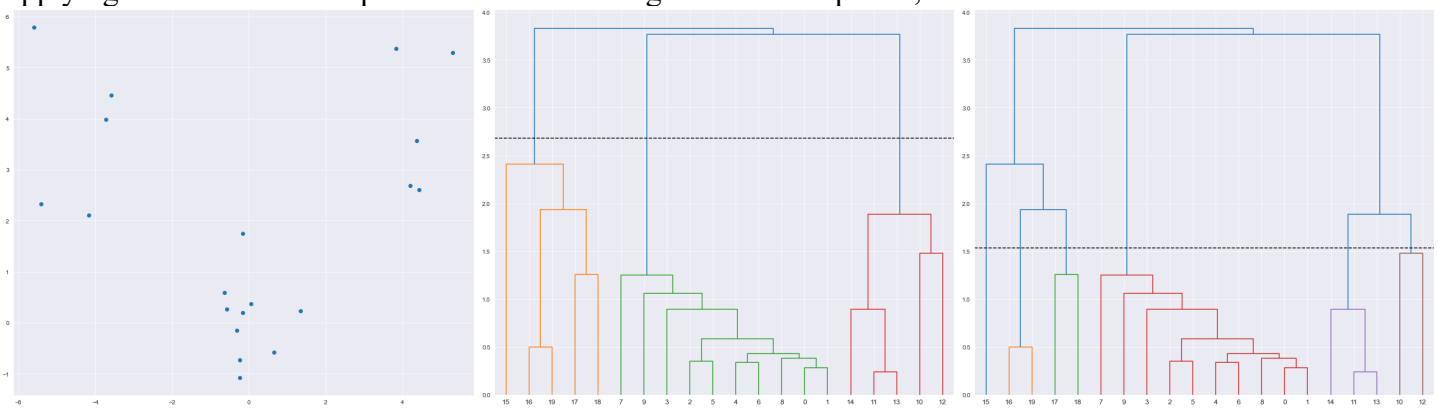
- The dendrogram illustrates the sample clustering as a function of d, with d increasing as you move up the chart
- A variety of clusterings can then be selected depending upon which distance you choose to make a cut point, that is to say, how high up the dendrogram we go

Rather than sklearn, we'll use scipy to perform hierarchical clustering in practice

- scipy keeps its hierarchical clustering functions/objects in the cluster.hierarchy module
- In particular, we will want dendrogram and linkage
    - linkage returns an array (known as the linkage matrix) that tracks every time that two clusters merge, which can then be visualized using dendrogram

- Once you've identified an appropriate cut point, you get the cluster label for your data using fcluster

Applying these to some sample data and illustrating different cut points, we can find:



As in *k*-means clustering, we could use an inertia/elbow plot or silhouette plots to help us determine what to set as our threshold value, but sometimes it is reasonable to just use a visual inspection

- Looking at this particular dendrogram a threshold that produces three clusters looks reasonable

There may also be other consideration that come from your particular application/problem

- For example, maybe you are working on a market segmentation problem for a business, and while additional market segments (clusters) might increase profit, they would also have an associated cost

Sample Code:

```
# import dendrogram and linkage from scipy
from scipy.cluster.hierarchy import dendrogram, linkage

# first we run linkage
Z = linkage(X, method='single')
# this returns a numpy array that we will now describe and print out
print( pd.DataFrame(Z,columns = ['cluster_1', 'cluster_2', 'distance', 'new_cluster_size']) )

# now plot the dendrogram
dendrogram(Z)
plt.show()

# need to import fcluster to get clusters for each point
from scipy.cluster.hierarchy import fcluster
# get the clusters, here using the default color_threshold value
fcluster(Z, t=0.7*max(Z[:,2]), criterion='distance')
```

Notes on the above code:

- We can control how linkage measures the distance d between clusters using the method argument
  - The default method is 'single', which finds the minimum distance between any pair of points between two clusters
  - Others include 'average', which finds the average distance between any pair of points between two clusters and 'centroid', which finds the distance between the centroids of two clusters
- There is an argument in dendrogram called color_threshold, with a default value of 0.7*max(Z[:,2]), that colors the clusters