Datacamp Notes
Heart B.

Classication

Process of predicting the class of given data points, gave discrete output

1. Supervised Learning

- Predict variables/features and a target variable
- Classification: Target variable consists of categories
- Regression: Target variable is continuous

2. Unsupervised Learning

- Uncovering hidden patterns from unlabeled data
- Example: Grouping customers into distinct categories (Clustering)

3. Reinforcement Learning

- Software agents interact with an environment
- Learn how to optimize their behavior
- Given a system of rewards and punishments
- Draws inspiration from behavioral psychology
- Applications: Economics, Genetics, Game Playing, AlphaGo

The Iris Dataset

One of the best examples for classification methods. These are the terms we should know

Exploratory Data Analysis (EDA)

Visualize the summarization of the data's main characteristics.

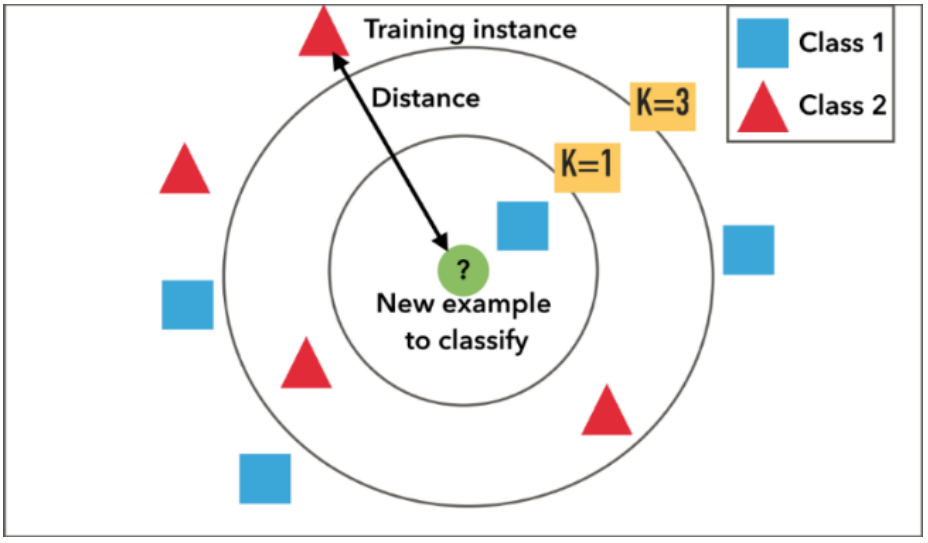
- df.head Print the first five rows
- pd.plotting.scatter matrix Visualize as graphs

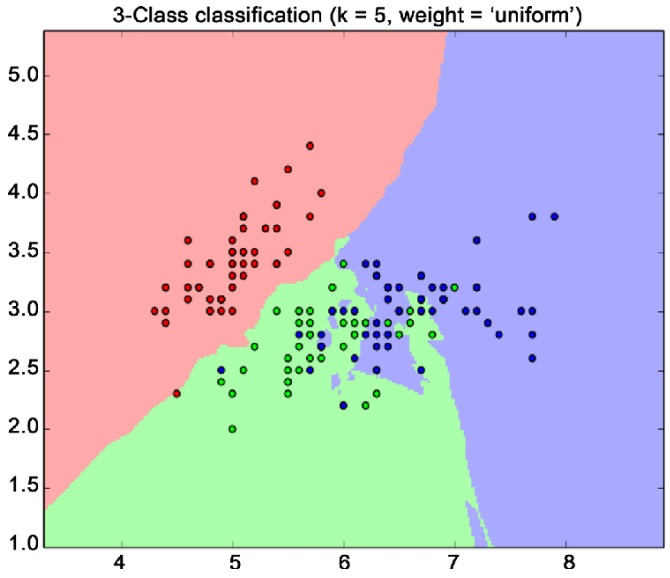
K-Nearest Neighbors

It's basically from the *Distance Function*

$$dist = \sqrt{\sum_{k=1}^{n}(p_k - q_k)^2}$$

Using *KNeighborsClassier* from *sklearn.neighbors*, and specify the *n_neighbors*





Codes:

```
# Import KNeighborsClassifier from sklearn.neighbors
from sklearn.neighbors import KNeighborsClassifier

# Create arrays for the features and the response variable
y = df['party'].values
X = df.drop('party', axis=1).values

# Create a k-NN classifier with 6 neighbors
knn = KNeighborsClassifier(n_neighbors=6)

# Fit the classifier to the data
knn.fit(X,y)
```

knn fit

```
# Import KNeighborsClassifier from sklearn.neighbors
from sklearn.neighbors import KNeighborsClassifier

# Create arrays for the features and the response variable
y = df['party'].values
X = df.drop('party', axis=1).values

# Create a k-NN classifier with 6 neighbors: knn
knn = KNeighborsClassifier(n_neighbors=6)

# Fit the classifier to the data
knn.fit(X,y)

# Predict the labels for the training data X
y_pred = knn.predict(X)

# Predict and print the label for the new data point X_new
new_prediction = knn.predict(X_new)
print("Prediction: {}".format(new_prediction))
```

knn fit and predict

Measuring model performance

We usually split data into training and test set using train test split from sklearn.model selection

- test size = 0.3, test 30%, train 70%
- **random state**
  - when a random state is set to an integer, train test split will return the same results for each execution.
  - when a random state is set to a None, train test split will return different results for each execution.
- stratify=y, stratify parameter will preserve the proportion of target as in original dataset, in the train and test datasets as well.
  For example, if there are 100 observations in the entire original dataset of which 80 are class a and 20 are class b and you set stratify = True, with a .7 : .3 train-test split, you will get a training set with 56 examples of class a and 14 examples of class b.

- **Fit/train** the classifier on the **training set** -> METHOD.fit(X_train, y_train)
- **Make predictions** on the **test set** -> METHOD.predict(X_test)
- Compare predictions with the known labels

```python
# Import necessary modules
from sklearn import datasets
import matplotlib.pyplot as plt

# Load the digits dataset: digits
digits = datasets.load_digits()

# Print the keys and DESCR of the dataset
print(digits.keys())
print(digits['DESCR'])

# Print the shape of the images and data keys
print(digits.images.shape)
print(digits.data.shape)

# Display digit 1010
plt.imshow(digits.images[1010], cmap=plt.cm.gray_r, interpolation='nearest')
plt.show()
```

The digits recognition datasets

```python
# Import necessary modules
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split

# Create feature and target arrays
X = digits.data
y = digits.target

# Split into training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=42, stratify=y)

# Create a k-NN classifier with 7 neighbors: knn
knn = KNeighborsClassifier(n_neighbors=7)

# Fit the classifier to the training data
knn.fit(X_train, y_train)

# Print the accuracy
print(knn.score(X_test, y_test))
```

Train/test split + Fit/predict/accuracy

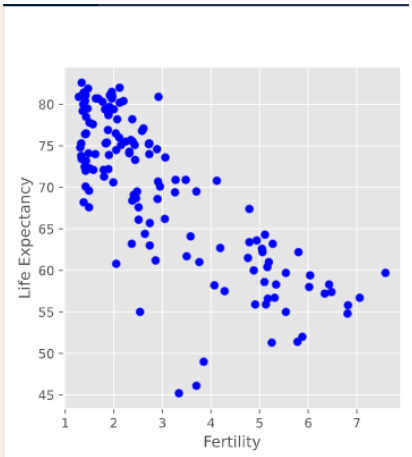
```
# Setup arrays to store train and test accuracies
neighbors = np.arange(1, 9)
train_accuracy = np.empty(len(neighbors))
test_accuracy = np.empty(len(neighbors))
# Loop over different values of k
for i, k in enumerate(neighbors):
    # Setup a k-NN Classifier with k neighbors: knn
    knn = KNeighborsClassifier(n_neighbors=k)
    # Fit the classifier to the training data
    knn.fit(X_train, y_train)
    #Compute accuracy on the training set
    train_accuracy[i] = knn.score(X_train, y_train)
    #Compute accuracy on the testing set
    test_accuracy[i] = knn.score(X_test, y_test)
# Generate plot
plt.title('k-NN: Varying Number of Neighbors')
plt.plot(neighbors, test_accuracy, label = 'Testing Accuracy')
plt.plot(neighbors, train_accuracy, label = 'Training Accuracy')
plt.legend()
plt.xlabel('Number of Neighbors')
plt.ylabel('Accuracy')
plt.show()
```

Overfitting and Underfitting

Regression

1. **Importing Data**
   - Import data using pandas pd.read csv('NAME.csv')
   - After we imported data to X and y, we need to reshape both to (-1,1) by X.reshape(-1,1)
   - sns.heatmap(df.corr(), square=True, cmap='RdYlGn')
     -> Green: Positively Correlated, Red: Negatively Correlated
2. **Fit & Predict for Regression**
   - Create the regressor: reg = LinearRegression()
   - Create the prediction space: prediction_space = np.linspace(min(X_fertility), max(X_fertility)).reshape(-1,1)
   - Fit the model: reg.t(X_fertility, y)
   - Compute predictions over the prediction space: y_pred = reg.predict(prediction_space)
   - Print R^2 score: print(reg.score(X_fertility, y))

```
# Import LinearRegression
from sklearn.linear_model import LinearRegression
# Create the regressor: reg
reg = LinearRegression()
# Create the prediction space
prediction_space = np.linspace(min(X_fertility), max(X_fertility))
.reshape(-1,1)
# Fit the model to the data
reg.fit(X_fertility, y)
# Compute predictions over the prediction space: y_pred
y_pred = reg.predict(prediction_space)
# Print R^2
print(reg.score(X_fertility, y))
# Plot regression line
plt.plot(prediction_space, y_pred, color='black', linewidth=3)
plt.show()
```



Fit and predict for regression

3. **Train/test split for regression**

   Here we use LinearRegression, mean_squared_error, train_test_split

   - Create training and test sets: X train, X test, y train, y test = train test split (X, y, test size = 0.3, random state=42)
   - Create the regressor: reg all = LinearRegression()
   - Fit the *regressor* to the training data: reg all.t(X train, y train)
   - Predict on the test data: y pred = reg all.predict(X test)

```
# Import necessary modules
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

# Create training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state=42)

# Create the regressor: reg_all
reg_all = LinearRegression()

# Fit the regressor to the training data
reg_all.fit(X_train, y_train)

# Predict on the test data: y_pred
y_pred = reg_all.predict(X_test)

# Compute and print R^2 and RMSE
print("R^2: {}".format(reg_all.score(X_test, y_test)))
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print("Root Mean Squared Error: {}".format(rmse))
```

Train/test split for regression

4. **k-Fold Cross-Validation**

   Cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample. It is a popular method because it is simple to understand and because it generally results in a less biased or less optimistic estimate of the model skill than other methods, such as a simple train/test split.
   We do this by **cross_val_score()**, don't forget to specify the cv 5-fold: cv=5

```python
# Import the necessary modules
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score

# Create a linear regression object: reg
reg = LinearRegression()

# Compute 5-fold cross-validation scores: cv_scores
cv_scores = cross_val_score(reg, X, y, cv=5)

# Print the 5-fold cross-validation scores
print(cv_scores)

print("Average 5-Fold CV Score: {}".format(np.mean(cv_scores)))
```

5-fold cross-validation

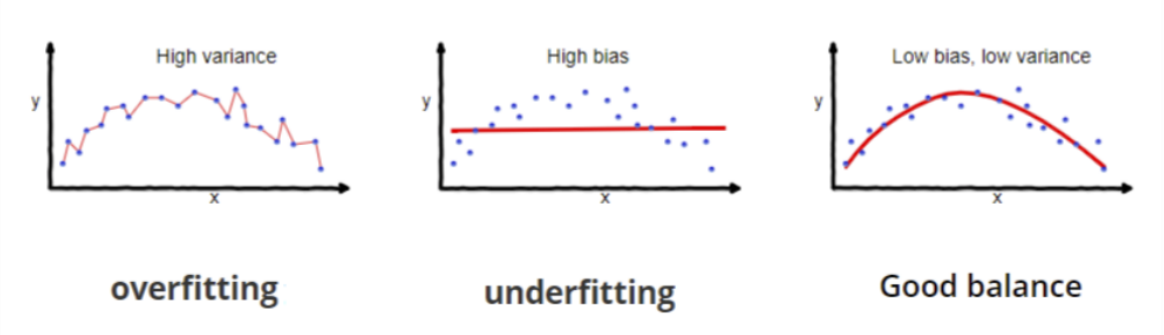
```python
# Import necessary modules
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score

# Create a linear regression object: reg
reg = LinearRegression()

# Perform 3-fold CV
cvscores_3 = cross_val_score(reg, X, y, cv=3)
print(np.mean(cvscores_3))

# Perform 10-fold CV
cvscores_10 = cross_val_score(reg, X, y, cv=10)
print(np.mean(cvscores_10))
```

k-fold CV comparison

5. **Regularization Regression**

Regularized regression is a type of regression where the coefficient estimates are constrained to zero. The magnitude (size) of coefficients, as well as the magnitude of the error term, are penalized. Complex models are discouraged, primarily to avoid overfitting.

- **Bias**: Biases are the underlying assumptions that are made by data to simplify the target function. Generalize data better, Less sensitive to single data points, Decreases training time
- **Variance**: Variance is a type of error that occurs due to a model's sensitivity to small fluctuations in the dataset. High variance of outliers



Lasso Regression

$$L_{lasso} = argmin_{\beta} \left(\|Y - \beta * X\|^2 + \lambda * \|\beta\|_1 \right)$$

Ridge Regression

$$L_{ridge} = argmin_{\beta} \left(\|Y - \beta * X\|^2 + \lambda * \|\beta\|_2^2 \right)$$

Ps. Watch https://youtu.be/Xm2C_gTAl8c

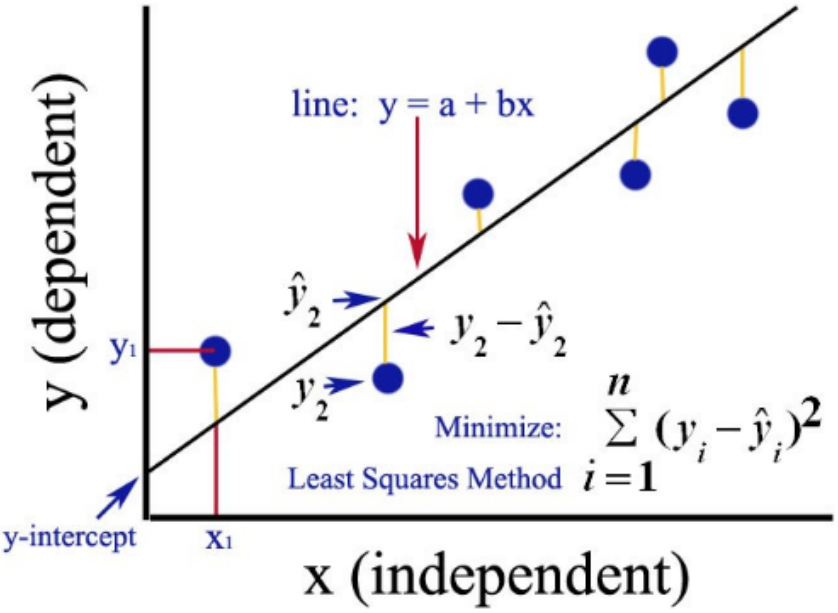
```
# Import Lasso
from sklearn.linear_model import Lasso

# Instantiate a lasso regressor: lasso
lasso = Lasso(alpha=0.4, normalize=True)

# Fit the regressor to the data
reg = lasso.fit(X, y)

# Compute and print the coefficients
lasso_coef = reg.coef_
print(lasso_coef)

# Plot the coefficients
plt.plot(range(len(df_columns)), lasso_coef)
plt.xticks(range(len(df_columns)), df_columns.values, rotation=60)
plt.margins(0.02)
plt.show()
```

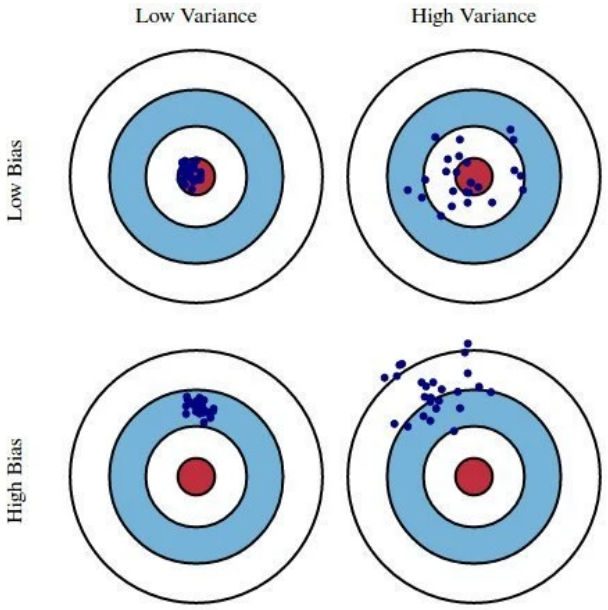
Lasso regression

```
# Import necessary modules
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score
# Setup the array of alphas and lists to store scores
alpha_space = np.logspace(-4, 0, 50)
ridge_scores = []
ridge_scores_std = []
# Create a ridge regressor: ridge
ridge = Ridge(normalize=True)
# Compute scores over range of alphas
for alpha in alpha_space:
    # Specify the alpha value to use: ridge.alpha
    ridge.alpha = alpha
    # Perform 10-fold CV: ridge_cv_scores
    ridge_cv_scores = cross_val_score(ridge, X, y, cv=10)
    # Append the mean of ridge_cv_scores to ridge_scores
    ridge_scores.append(np.mean(ridge_cv_scores))
    # Append the std of ridge_cv_scores to ridge_scores_std
    ridge_scores_std.append(np.std(ridge_cv_scores))
# Display the plot
display_plot(ridge_scores, ridge_scores_std)
```

Ridge regression

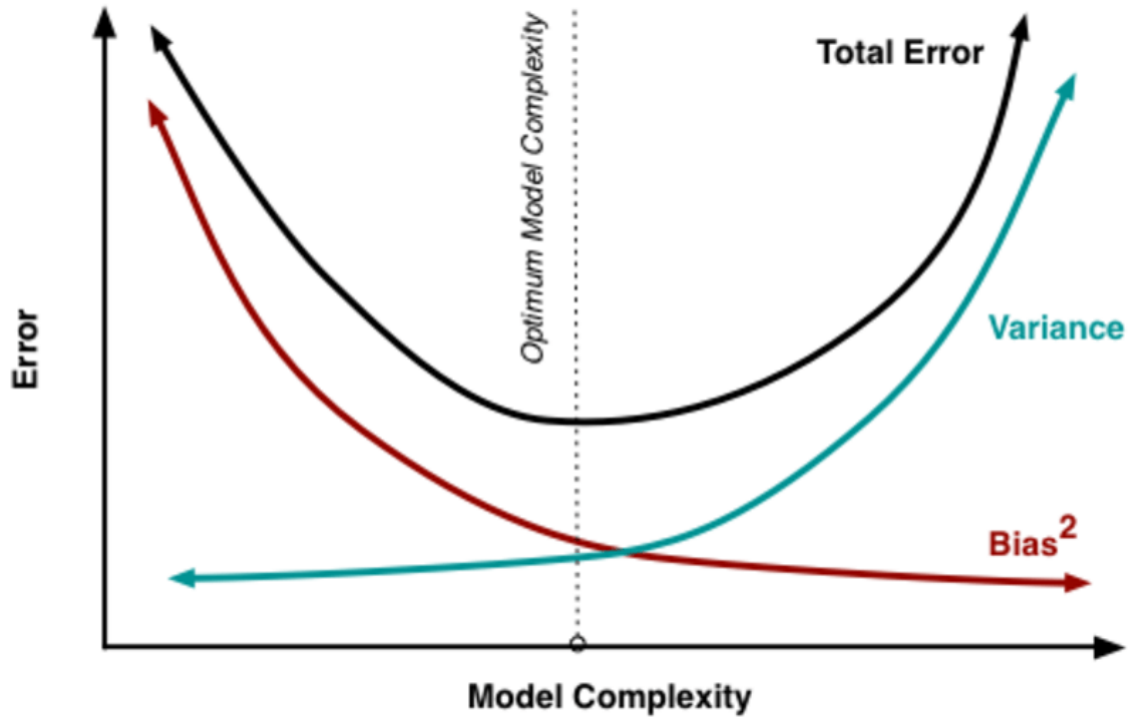
We basically need the **Ordinary Least Squares (OLS)**






```python
# Import necessary modules
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

# Create training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)

# Instantiate a k-NN classifier: knn
knn = KNeighborsClassifier(n_neighbors=6)

# Fit the classifier to the training data
knn.fit(X_train, y_train)

# Predict the labels of the test data: y_pred
y_pred = knn.predict(X_test)

# Generate the confusion matrix and classification report
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

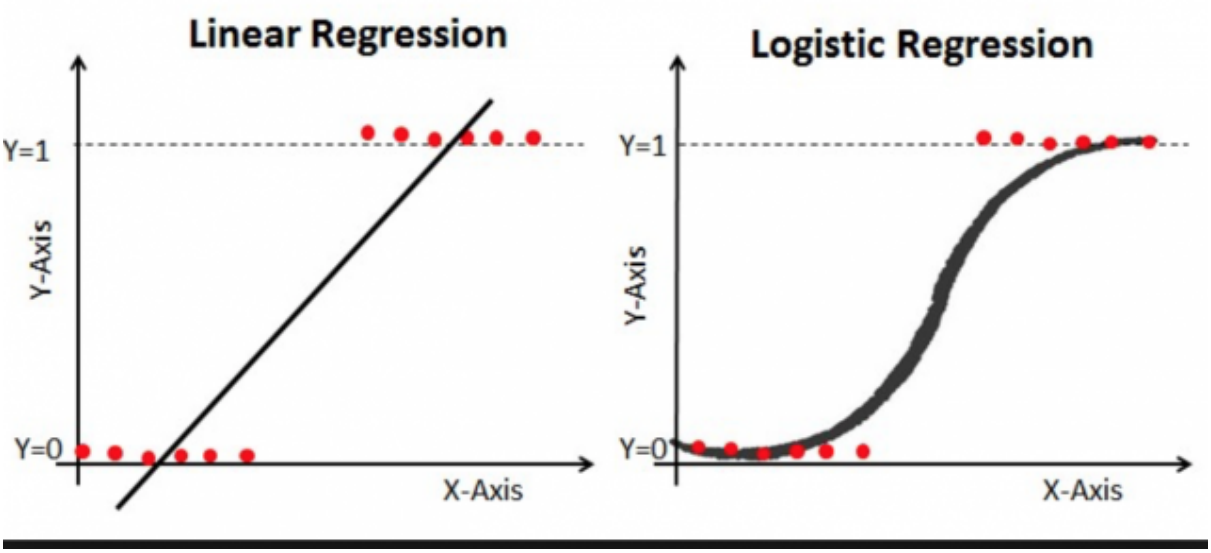
Metrics for classification



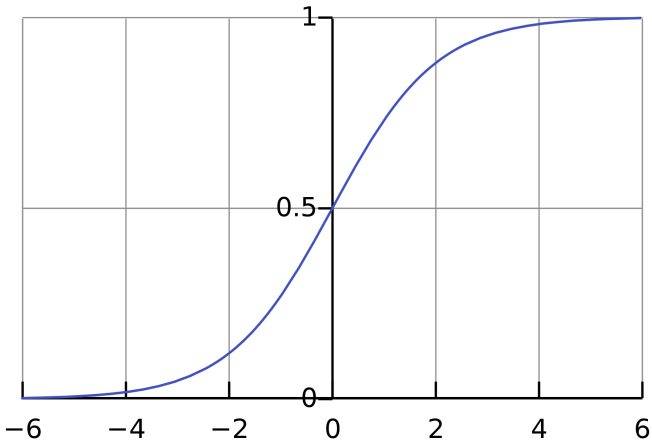
The model complexity has to be at the appropriate level

Fine Tuning the model

Predict probability between 2 choices, e.g., 0 and 1



Standard Logistic function



$$\sigma(t) = \frac{e^t}{e^t + 1} = \frac{1}{1 + e^{-t}}$$

```python
# Import the necessary modules
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, classification_report

# Create training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.4, random_state=42)

# Create the classifier: logreg
logreg = LogisticRegression()

# Fit the classifier to the training data
logreg.fit(X_train, y_train)

# Predict the labels of the test set: y_pred
y_pred = logreg.predict(X_test)

# Compute and print the confusion matrix and classification report
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

Logistic regression model

ROC Curve

In logistic regression, if we move the “decision threshold”, e.g., 0.5 to other values. The “correct value” in the **confusion matrix** will change.

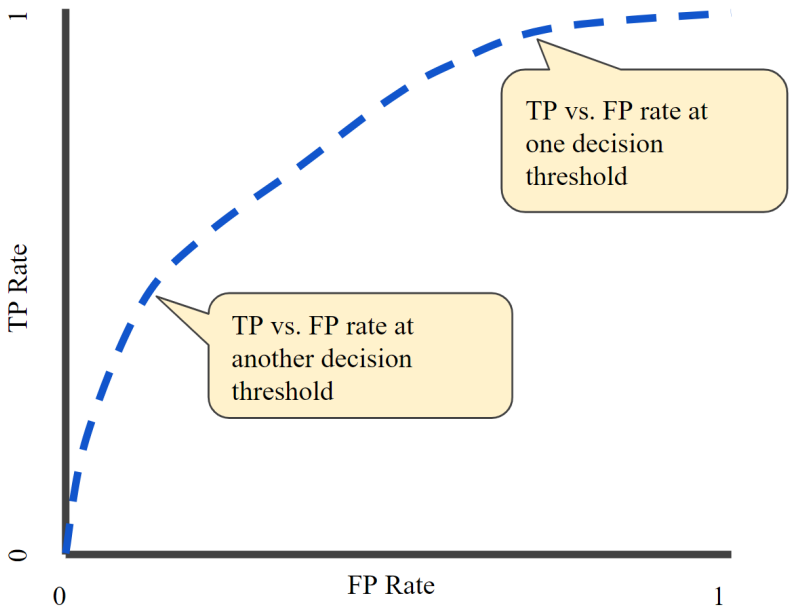
Acquired Class	True Class T	True Class F
Y	True Positives (TP)	False Positives (FP)
N	False Negatives (FN)	True Negatives (TN)

$$\text{True Positive Rate (TPR)} = \frac{TP}{TP + FN}$$

$$\text{False Positive Rate (FPR)} = \frac{FP}{FP + TN}$$

$$\text{Accuracy (ACC)} = \frac{TP + TN}{TP + FP + TN + FN}$$

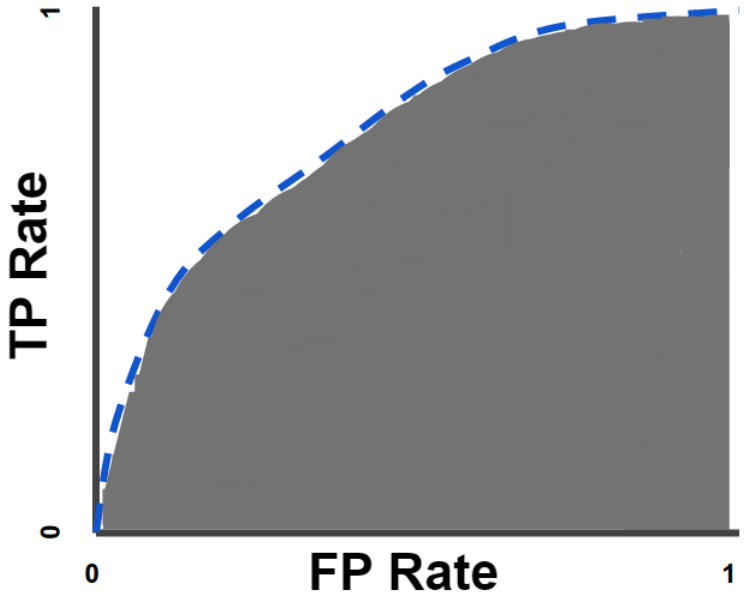
Then, we will use the “True positive rate” and “False positive rate” to plot the ROC curve




```
# Import necessary modules
from sklearn.metrics import roc_curve

# Compute predicted probabilities: y_pred_prob
y_pred_prob = logreg.predict_proba(X_test)[:,1]

# Generate ROC curve values: fpr, tpr, thresholds
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

# Plot ROC curve
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()
```

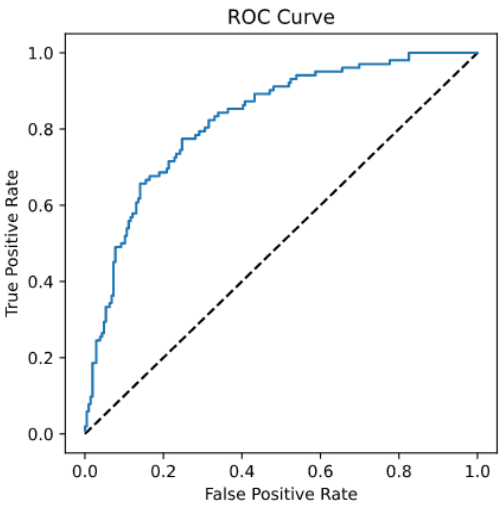
Plotting an ROC curve

AUC: Area under the ROC curve



AUC values range from 0 to 1. One whose predictions are 100% correct has an AUC of 1.0.

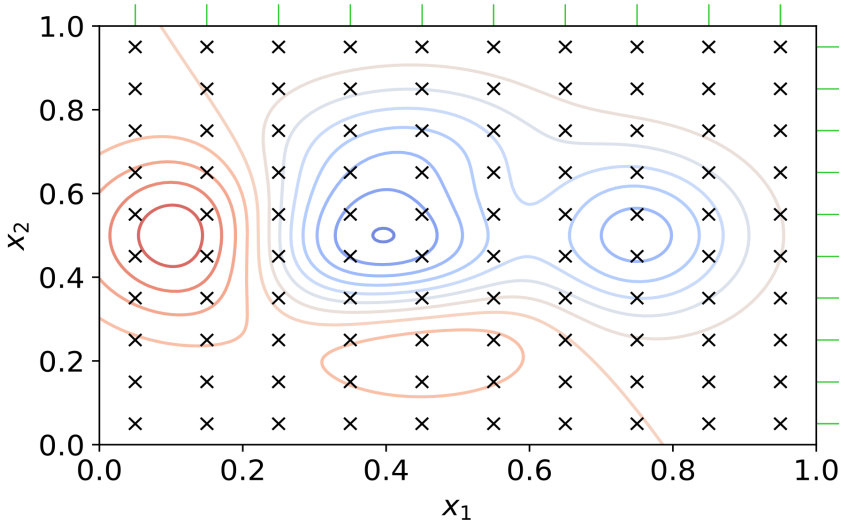
```
# Import necessary modules
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import
cross_val_score

# Compute predicted probabilities: y_pred_prob
y_pred_prob = logreg.predict_proba(X_test)[:,1]

# Compute and print AUC score
print("AUC: {}".format(roc_auc_score(y_test,
y_pred_prob)))

# Compute cross-validated AUC scores: cv_auc
cv_auc = cross_val_score(logreg, X, y, cv=5,
scoring='roc_auc')

# Print list of AUC scores
print("AUC scores computed using 5-fold
cross-validation: {}".format(cv_auc))
```



AUC computation

Hyperparameter tuning
The problem of choosing a set of optimal hyperparameters for a learning algorithm. A hyperparameter is a parameter whose value is used to control the learning process. By contrast, the values of other parameters (typically node weights) are learned.

GridSearchCV (Grid search, parameter sweep -> “exhaustive searching” (brute-force))
A grid search algorithm must be guided by some performance metric, typically measured by cross-validation on the training set or evaluation on a held-out validation set.



Grid Search ‘x’ is lined as a grid

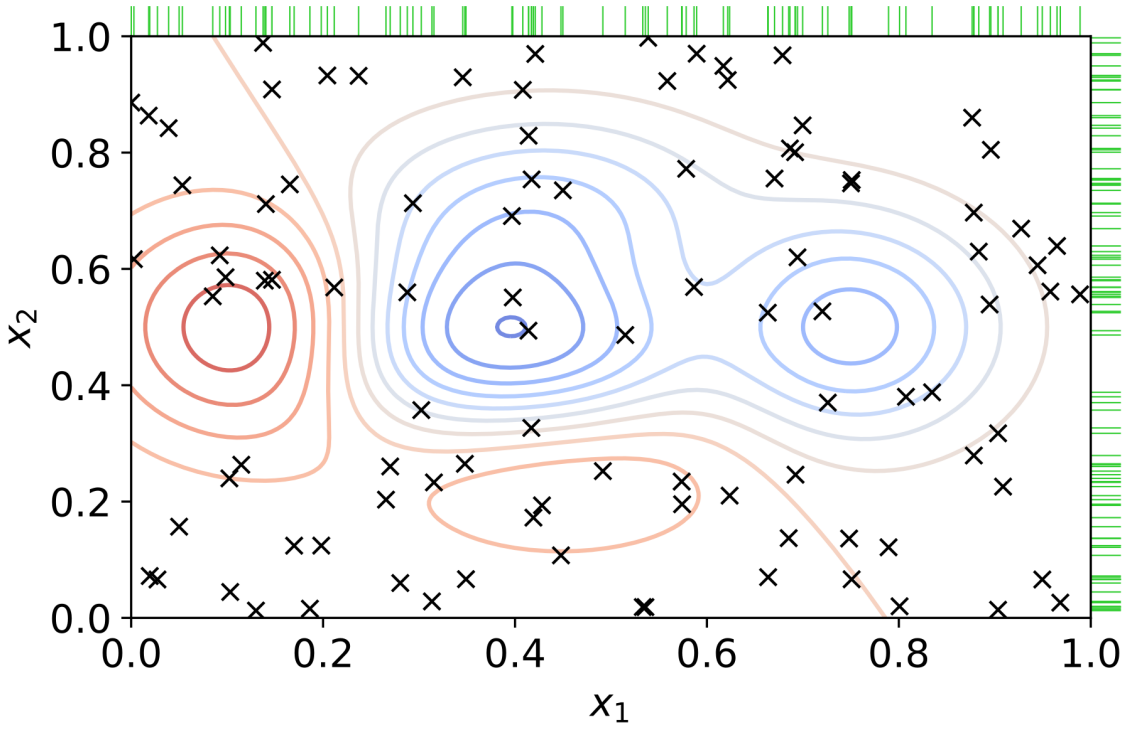
```python
# Import necessary modules
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

# Setup the hyperparameter grid
c_space = np.logspace(-5, 8, 15)
param_grid = {'C': c_space}

# Instantiate a logistic regression classifier: logreg
logreg = LogisticRegression()

# Instantiate the GridSearchCV object: logreg_cv
logreg_cv = GridSearchCV(logreg, param_grid, cv=5)

# Fit it to the data
logreg_cv.fit(X,y)

# Print the tuned parameters and score
print("Tuned Logistic Regression Parameters: {}".format(logreg_cv.best_params_))
print("Best score is {}".format(logreg_cv.best_score_))
```

Grid search parameter tuning

RandomizedSearchCV replaces the exhaustive enumeration of all combinations by selecting them randomly. This can be simply applied to the discrete setting described above, but also generalizes to continuous and mixed spaces. ***It can outperform Grid search***, especially ***when only a small number of hyperparameters*** affects the final performance of the machine learning algorithm.

```
# Import necessary modules
from scipy.stats import randint
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import RandomizedSearchCV

# Setup the parameters and distributions to sample from: param_dist
param_dist = {"max_depth": [3, None],
              "max_features": randint(1, 9),
              "min_samples_leaf": randint(1, 9),
              "criterion": ["gini", "entropy"]}
# Instantiate a Decision Tree classifier: tree
tree = DecisionTreeClassifier()
# Instantiate the RandomizedSearchCV object: tree_cv
tree_cv = RandomizedSearchCV(tree, param_dist, cv=5)
# Fit it to the data
tree_cv.fit(X,y)
# Print the tuned parameters and score
print("Tuned Decision Tree Parameters: {}".format(tree_cv.best_params_))
print("Best score is {}".format(tree_cv.best_score_))
```

Randomized Search parameter tuning



Randomized Search ‘x’

Both GridSearchCV and RandomizedSearchCV are ***embarrassingly parallel***.

Training -> Validation -> Hold-out

Hold-out Sometimes referred to as "**testing**" data, a holdout subset provides a final estimate of the machine learning model's performance after it has been trained and validated. Holdout sets should never be used to make decisions about which algorithms to use or for improving or tuning algorithms.

Hold-out set: Classification

```
# Import necessary modules
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
# Create the hyperparameter grid
c_space = np.logspace(-5, 8, 15)
param_grid = {'C': c_space, 'penalty': ['l1', 'l2']}
# Instantiate the logistic regression classifier: logreg
logreg = LogisticRegression()
# Create train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)
# Instantiate the GridSearchCV object: logreg_cv
logreg_cv = GridSearchCV(logreg, param_grid, cv=5)
# Fit it to the training data
logreg_cv.fit(X_train, y_train)
# Print the optimal parameters and best score
print("Tuned Logistic Regression Parameter: {}".format(logreg_cv.best_params_))
print("Tuned Logistic Regression Accuracy: {}".format(logreg_cv.best_score_))
```

Hold-out set: Regression

```
# Import necessary modules
from sklearn.linear_model import ElasticNet
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
# Create train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)
# Create the hyperparameter grid
l1_space = np.linspace(0, 1, 30)
param_grid = {'l1_ratio': l1_space}
# Instantiate the ElasticNet regressor: elastic_net
elastic_net = ElasticNet()
# Setup the GridSearchCV object: gm_cv
gm_cv = GridSearchCV(elastic_net, param_grid, cv=5)
# Fit it to the training data
gm_cv.fit(X_train, y_train)
# Predict on the test set and compute metrics
y_pred = gm_cv.predict(X_test)
r2 = gm_cv.score(X_test, y_test)
mse = mean_squared_error(y_test, y_pred)
print("Tuned ElasticNet l1 ratio: {}".format(gm_cv.best_params_))
print("Tuned ElasticNet R squared: {}".format(r2))
print("Tuned ElasticNet MSE: {}".format(mse))
```

It use ElasticNet regularization: combined the strength of Ridge and Lasso regression

$$\frac{\sum_{i=1}^{n}(y_i - x_i^J \hat{\beta})^2}{2n} + \lambda \left(\frac{1-\alpha}{2} \sum_{j=1}^{m} \hat{\beta}_j^2 + \alpha \sum_{j=1}^{m} |\hat{\beta}_j| \right)$$

It used penalties from L1 and L2 regularization

L1 Regularization

$$\text{Cost} = \sum_{i=0}^{N}(y_i - \sum_{j=0}^{M} x_{ij}W_j)^2 + \lambda \sum_{j=0}^{M} |W_j|$$

L2 Regularization

$$\text{Cost} = \sum_{i=0}^{N}(y_i - \sum_{j=0}^{M} x_{ij}W_j)^2 + \lambda \sum_{j=0}^{M} W_j^2$$

Loss function

Regularization Term

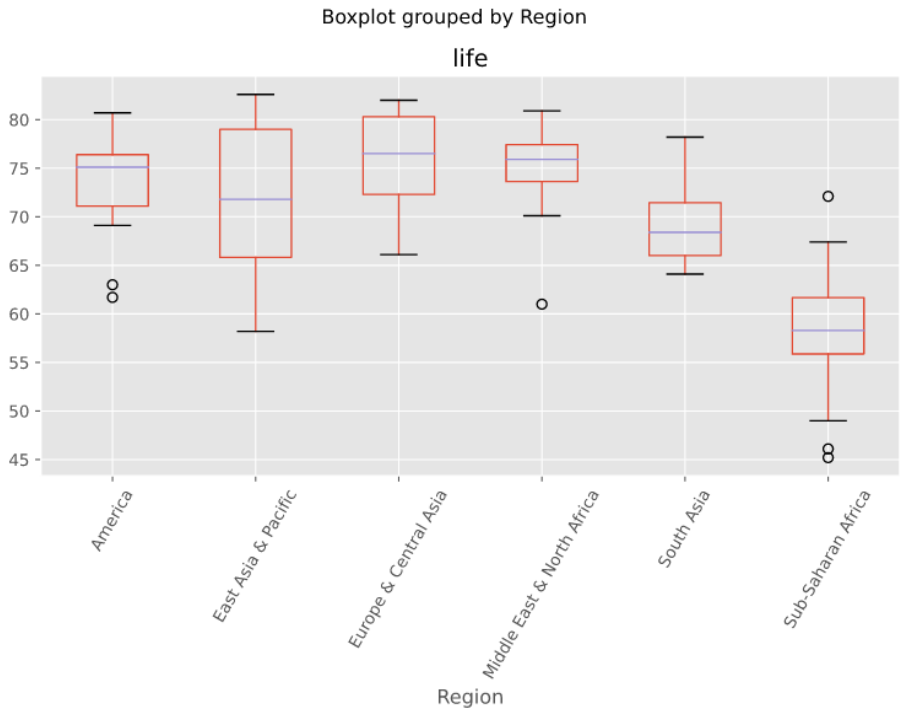
The main intuitive difference between the L1 and L2 regularization is that L1 regularization tries to estimate the median of the data while the L2 regularization tries to estimate the mean of the data to avoid overfitting.

Preprocessing Data

Convert “label” to 0, 1 form (convert label into dummy)
scikit-learn: OneHotEncoder()
pandas: get_dummies()

Using pandas to read csv file and plot the boxplot (x,y)=(region,life)

```
# Import pandas
import pandas as pd
# Read 'gapminder.csv' into a DataFrame: df
df = pd.read_csv('gapminder.csv')
# Create a boxplot of life expectancy per region
df.boxplot('life', 'Region', rot=60)
# Show the plot
plt.show()
```



```
# Create dummy variables: df_region
df_region = pd.get_dummies(df)
# Print the columns of df_region
print(df_region.columns)
# Create dummy variables with drop_first=True: df_region
df_region = pd.get_dummies(df, drop_first=True)
# Print the new columns of df_region
print(df_region.columns)
```

get_dummies but drop the first variables of x-axis

```
# Import necessary modules
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score
# Instantiate a ridge regressor: ridge
ridge = Ridge(alpha=0.5, normalize=True)
# Perform 5-fold cross-validation: ridge_cv
ridge_cv = cross_val_score(ridge, X, y, cv=5)
# Print the cross-validated scores
print(ridge_cv)
```

Regression with categorical features

Handling missing data

Dropping the missing data -> df.dropna()

Strategies (Inputting missing data) -> Ex. use the mean of non-missing data (sklearn's Imputer),

Inputting with pipeline (sklearn's Pipeline) -> Imputer -> Some regression (e.g. logreg)

```
# Convert '?' to NaN
df[df == '?'] = np.nan
# Print the number of NaNs
print(df.isnull().sum())
# Print shape of original DataFrame
print("Shape of Original DataFrame: {}".format(df.shape))
# Drop missing values and print shape of new DataFrame
df = df.dropna()
# Print shape of new DataFrame
print("Shape of DataFrame After Dropping All Rows with Missing Values: {}".format(df.shape))
```

Replace '?' with NaN, check the shape, then drop NaN, check the shape again

```
# Import the Imputer module
from sklearn.preprocessing import Imputer
from sklearn.svm import SVC
# Setup the Imputation transformer: imp
imp = Imputer(missing_values='NaN', strategy='most_frequent', axis=0)
# Instantiate the SVC classifier: clf
clf = SVC()
# Setup the pipeline with the required steps: steps
steps = [('imputation', imp),
         ('SVM', clf)]
```

Set up imputer with strategy, SVC = Support Vector Machine -> draw a decent line that classifies the different class of data

```
# Import necessary modules
from sklearn.preprocessing import Imputer
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
# Setup the pipeline steps: steps
steps = [('imputation', Imputer(missing_values='NaN', strategy='most_frequent', axis=0)),
         ('SVM', SVC())]
# Create the pipeline: pipeline
pipeline = Pipeline(steps)
# Create training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Fit the pipeline to the train set
pipeline.fit(X_train, y_train)
# Predict the labels of the test set
y_pred = pipeline.predict(X_test)
# Compute metrics
print(classification_report(y_test, y_pred))
```

Preprocess data -> fit -> predict, Report: scores on each class

Centering and scaling

Scale -> Features on larger scale can unduly influence the model

Want features to be on similar scale

Normalizing

Normalizing -> Standardization data (subtract mean, divided by variance)

All feature centered around zero, variance one

Min = Zero, Max = one

Normalize data range from -1 to +1

sklearn -> scale (sklearn.preprocessing)

pipeline -> StandardScaler

```
# Import scale
from sklearn.preprocessing import scale
# Scale the features: X_scaled
X_scaled = scale(X)
# Print the mean and standard deviation of the unscaled features
print("Mean of Unscaled Features: {}".format(np.mean(X)))
print("Standard Deviation of Unscaled Features: {}".format(np.std(X)))
# Print the mean and standard deviation of the scaled features
print("Mean of Scaled Features: {}".format(np.mean(X_scaled)))
print("Standard Deviation of Scaled Features: {}".format(np.std(X_scaled)))
```

Centralizing the data

```
# Import the necessary modules
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# Setup the pipeline steps: steps
steps = [('scaler', StandardScaler()),
         ('knn', KNeighborsClassifier())]
# Create the pipeline: pipeline
pipeline = Pipeline(steps)
# Create train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Fit the pipeline to the training set: knn_scaled
knn_scaled = pipeline.fit(X_train, y_train)
# Instantiate and fit a k-NN classifier to the unscaled data
knn_unscaled = KNeighborsClassifier().fit(X_train, y_train)
# Compute and print metrics
print('Accuracy with Scaling: {}'.format(knn_scaled.score(X_test, y_test)))
print('Accuracy without Scaling: {}'.format(knn_unscaled.score(X_test, y_test)))
```

Compare the accuracy between before and after scaling

<script.py> output:
Accuracy with Scaling: 0.7700680272108843
Accuracy without Scaling: 0.6979591836734694
You can see that we got better accuracy after scaling the data

Specify the hyperparameter space using the following notation: 'step_name__parameter_name'. Here, the step_name is SVM, and the parameter_names are C and gamma.

```
# Setup the pipeline
steps = [('scaler', StandardScaler()),
         ('SVM', SVC())]
pipeline = Pipeline(steps)
# Specify the hyperparameter space
parameters = {'SVM__C':[1, 10, 100],
              'SVM__gamma':[0.1, 0.01]}
# Create train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=21)
# Instantiate the GridSearchCV object: cv
cv = GridSearchCV(pipeline, parameters, cv=3)
# Fit to the training set
cv.fit(X_train, y_train)
# Predict the labels of the test set: y_pred
y_pred = cv.predict(X_test)
# Compute and print metrics
print("Accuracy: {}".format(cv.score(X_test, y_test)))
print(classification_report(y_test, y_pred))
print("Tuned Model Parameters: {}".format(cv.best_params_))
```

```
# Setup the pipeline steps: steps
steps = [('imputation', Imputer(missing_values='NaN', strategy='mean', axis=0)),
         ('scaler', StandardScaler()),
         ('elasticnet', ElasticNet())]
# Create the pipeline: pipeline
pipeline = Pipeline(steps)
# Specify the hyperparameter space
parameters = {'elasticnet__l1_ratio':np.linspace(0,1,30)}
# Create train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)
# Create the GridSearchCV object: gm_cv
gm_cv = GridSearchCV(pipeline, parameters, cv=3)
# Fit to the training set
gm_cv.fit(X_train, y_train)
# Compute and print the metrics
r2 = gm_cv.score(X_test, y_test)
print("Tuned ElasticNet Alpha: {}".format(gm_cv.best_params_))
print("Tuned ElasticNet R squared: {}".format(r2))
```

Unsupervised Learning

Unsupervised is a machine learning technique in which the users do not need to supervise the model. Instead, it allows the model to work on its own to discover patterns and information that was previously undetected. It mainly deals with the unlabelled data.

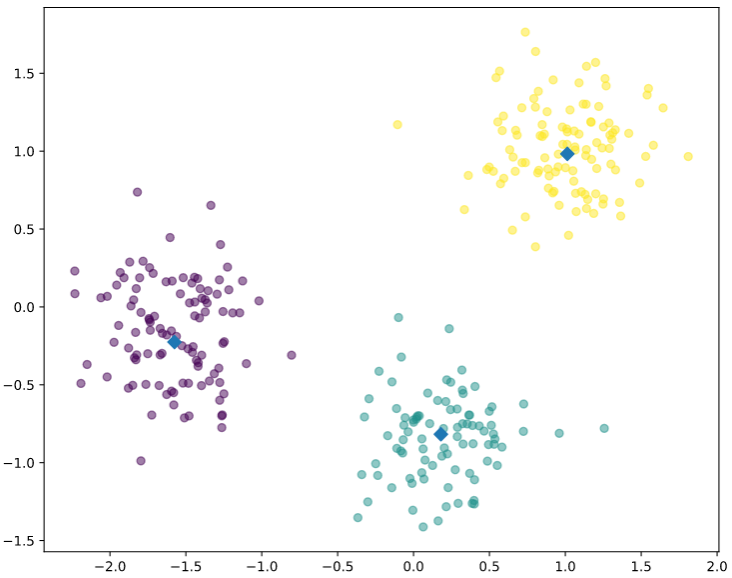
KMeans clustering

```
# Import KMeans
from sklearn.cluster import KMeans
# Create a KMeans instance with 3 clusters: model
model = KMeans(n_clusters=3)
# Fit model to points
model.fit(points)
# Determine the cluster labels of new_points: labels
labels = model.predict(new_points)
# Print cluster labels of new_points
print(labels)
```

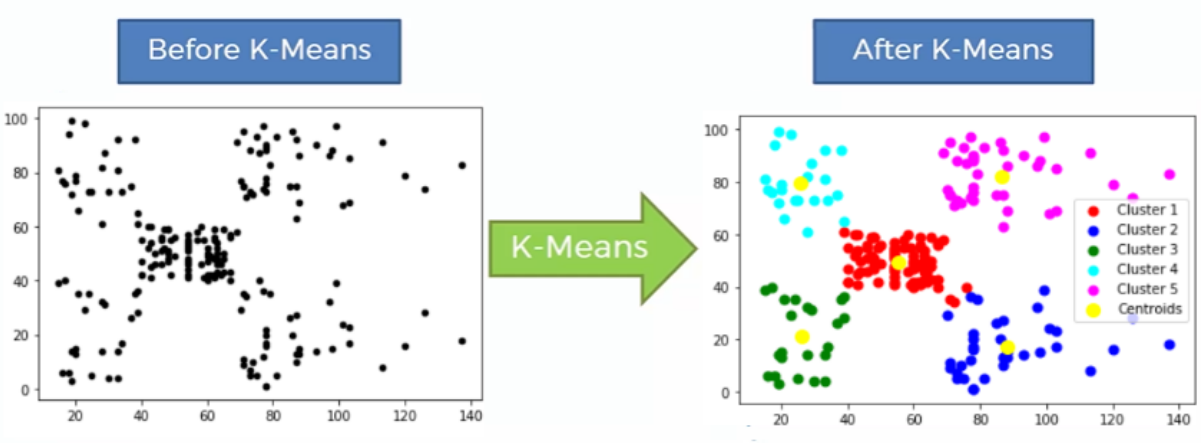
Classificate new_points by using KMeans clustering

Finding the centroid of the clusters (each cluster)

```
# Import pyplot
import matplotlib.pyplot as plt
# Assign the columns of new_points: xs and ys
xs = new_points[:,0]
ys = new_points[:,1]
# Make a scatter plot of xs and ys, using labels to define the
colors
plt.scatter(xs,ys,c=labels,alpha=0.5)
# Assign the cluster centers: centroids
centroids = model.cluster_centers_
# Assign the columns of centroids: centroids_x, centroids_y
centroids_x = centroids[:,0]
centroids_y = centroids[:,1]
# Make a scatter plot of centroids_x and centroids_y
plt.scatter(centroids_x, centroids_y,marker='D', s=50)
plt.show()
```

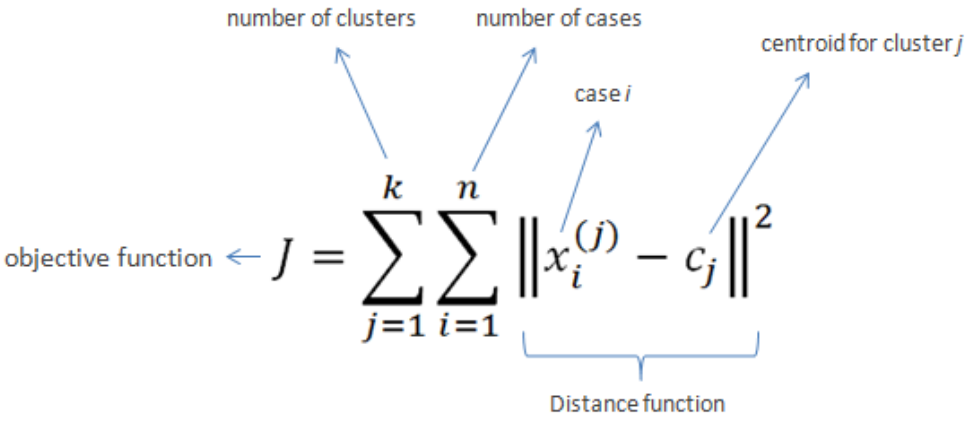


Output from finding centroids (and separate clusters by color; c=labels)



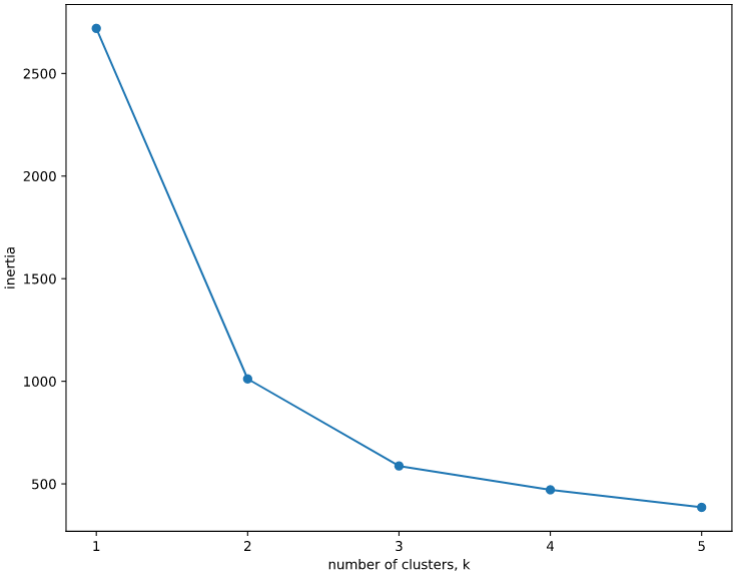
Before and after applying KMeans with centroids as well

The K-means algorithm aims to choose centroids that **minimise the inertia**, or within-cluster sum-of-squares criterion:




```python
ks = range(1, 6)
inertias = []

for k in ks:
    # Create a KMeans instance with k clusters:
model
    model = KMeans(n_clusters=k)

    # Fit model to samples
    model.fit(samples)

    # Append the inertia to the list of inertias
    inertias.append(model.inertia_)

# Plot ks vs inertias
plt.plot(ks, inertias, '-o')
plt.xlabel('number of clusters, k')
plt.ylabel('inertia')
plt.xticks(ks)
plt.show()
```

Comparing the inertia (lower is better) and number of clusters



5 clusters is obviously better than 1 cluster (look at inertia number)

```
# Create a KMeans model with 3 clusters: model
model = KMeans(n_clusters=3)
# Use fit_predict to fit model and obtain cluster labels: labels
labels = model.fit_predict(samples)
# Create a DataFrame with labels and varieties as columns: df
df = pd.DataFrame({'labels': labels, 'varieties': varieties})
# Create crosstab: ct
ct = pd.crosstab(df['labels'], df['varieties'])
# Display ct
print(ct)
```

Evaluating the clusters (look at ct output)

<script.py> output:

varieties	Canadian wheat	Kama wheat	Rosa wheat
labels			
0	0	1	60
1	68	9	0
2	2	60	10

pd.crosstab(): Compute a simple cross tabulation of two (or more) factors. By default computes a frequency table of the factors unless an array of values and an aggregation function are passed. (Cross between labels and varieties)

Transforming features for better clustering (preprocessing)

Standard scaler - transform each feature to have mean = 0, and variance = 1
“StandardScaler of sklearn”
StandardScaler -> KMeans (can be combined using Pipeline)

```
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
scaler = StandardScaler()
kmeans = KMeans(n_clusters=3)
from sklearn.pipeline import make_pipeline
pipeline = make_pipeline(scaler, kmeans)
pipeline.fit(samples)
```

```
Pipeline(steps=...)
```

```
labels = pipeline.predict(samples)
```

Example of StandardScaler and KMeans pipeline

```
# Import pandas
import pandas as pd
# Fit the pipeline to samples
pipeline.fit(samples)
# Calculate the cluster labels: labels
labels = pipeline.predict(samples)
# Create a DataFrame with labels and species as columns: df
df = pd.DataFrame({'labels':labels,'species':species})
# Create crosstab: ct
ct = pd.crosstab(df['labels'], df['species'])
# Display ct
print(ct)
```

Pipeline used and created pd.DataFrame from dictionary

```
# Import Normalizer
from sklearn.preprocessing import Normalizer
# Create a normalizer: normalizer
normalizer = Normalizer()
# Create a KMeans model with 10 clusters: kmeans
kmeans = KMeans(n_clusters=10)
# Make a pipeline chaining normalizer and kmeans: pipeline
pipeline = make_pipeline(normalizer, kmeans)
# Fit pipeline to the daily price movements
pipeline.fit(movements)
```

Normalize before K-Means is another method than standardization

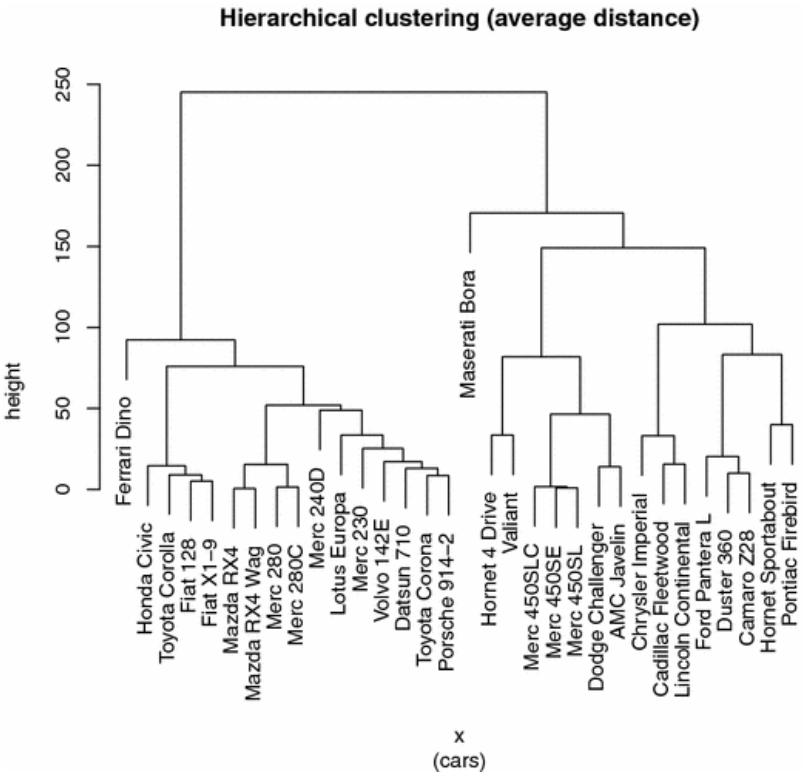
Normalization typically means rescales the values into a range of [0,1].
Standardization typically means rescales data to have a mean of 0 and a standard deviation of 1 (unit variance).

```
# Import pandas
import pandas as pd
# Predict the cluster labels: labels
labels = pipeline.predict(movements)
# Create a DataFrame aligning labels and companies: df
df = pd.DataFrame({'labels': labels, 'companies': companies})
# Display df sorted by cluster label
print(df.sort_values('labels'))
```

Print the sorted-by-labels table after applying pipeline to the raw data

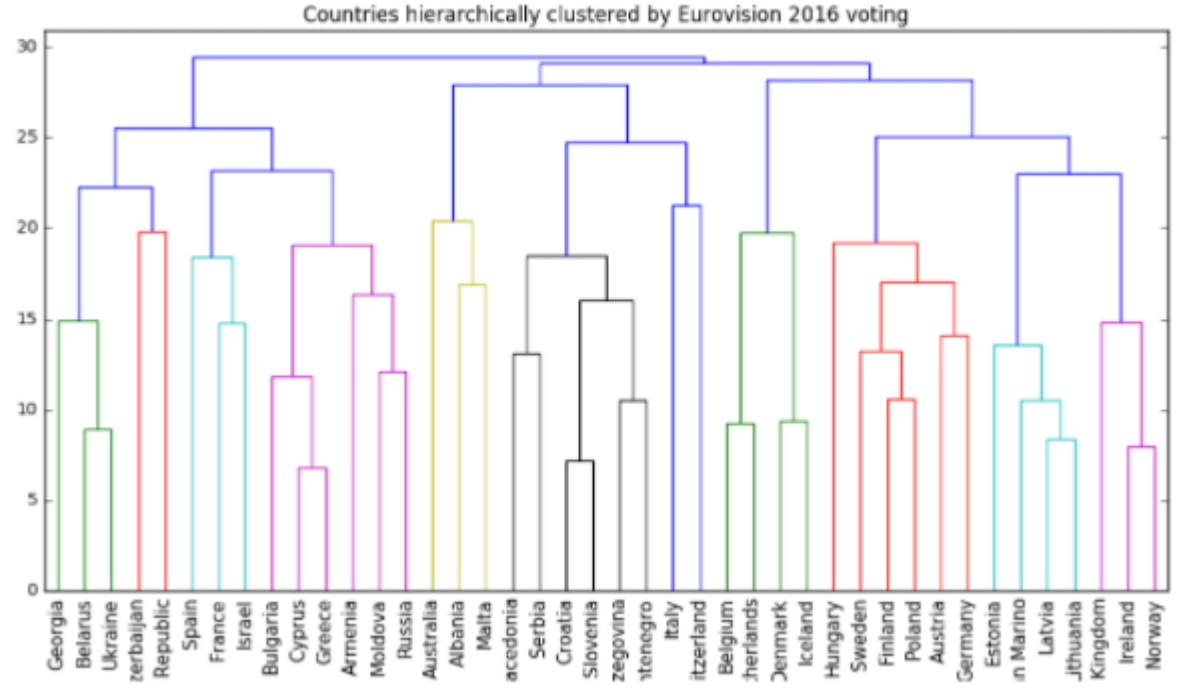
Hierarchical clustering

is an algorithm that groups similar objects into groups called clusters. The endpoint is a set of clusters, where each cluster is distinct from each other cluster, and the objects within each cluster are broadly similar to each other. “**Dendrograms**”



Example of hierarchical clustering (cars specs)

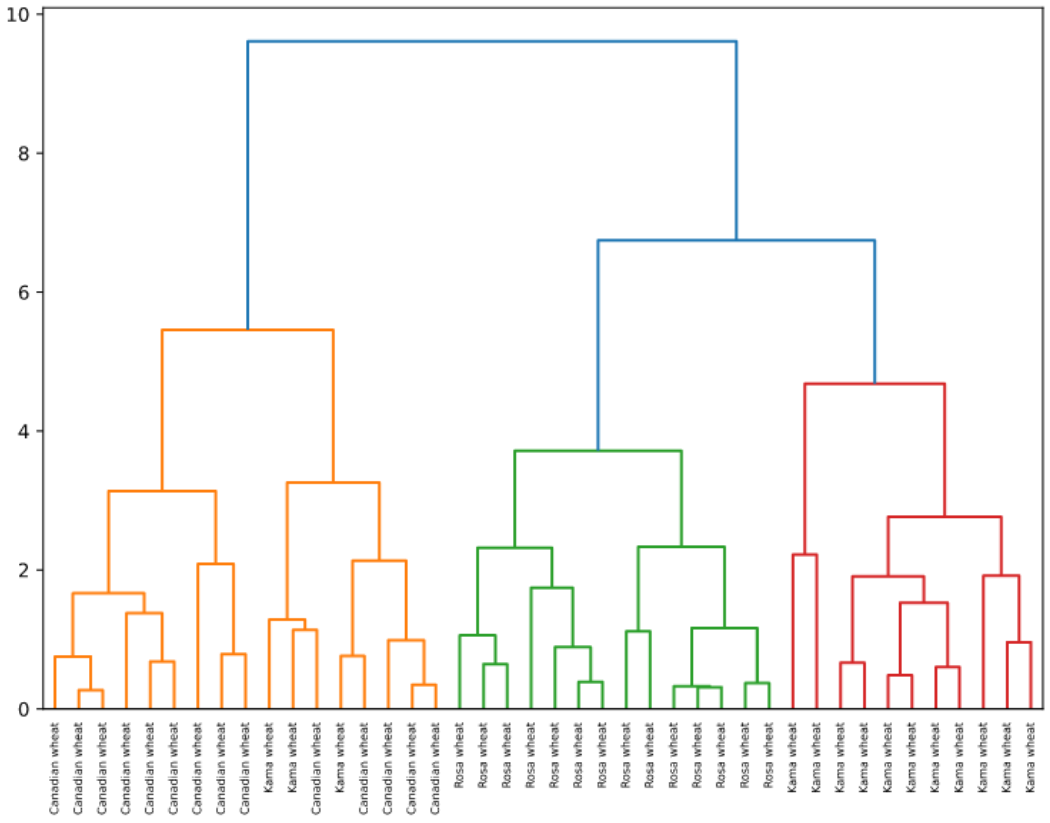
Agglomerative Hierarchical Clustering - At each step, the two closest clusters are merged. Then continue until all classes are in a single cluster. (Read from bottom up)



Example of agglomerative hierarchical clustering (countries) (vertical line = clusters)

```
# Perform the necessary imports
from scipy.cluster.hierarchy import linkage,
dendrogram
import matplotlib.pyplot as plt
# Calculate the linkage: mergings
mergings = linkage(samples, method='complete')
# Plot the dendrogram, using varieties as labels
dendrogram(mergings,
           labels=varieties,
           leaf_rotation=90,
           leaf_font_size=6,
)
plt.show()
```

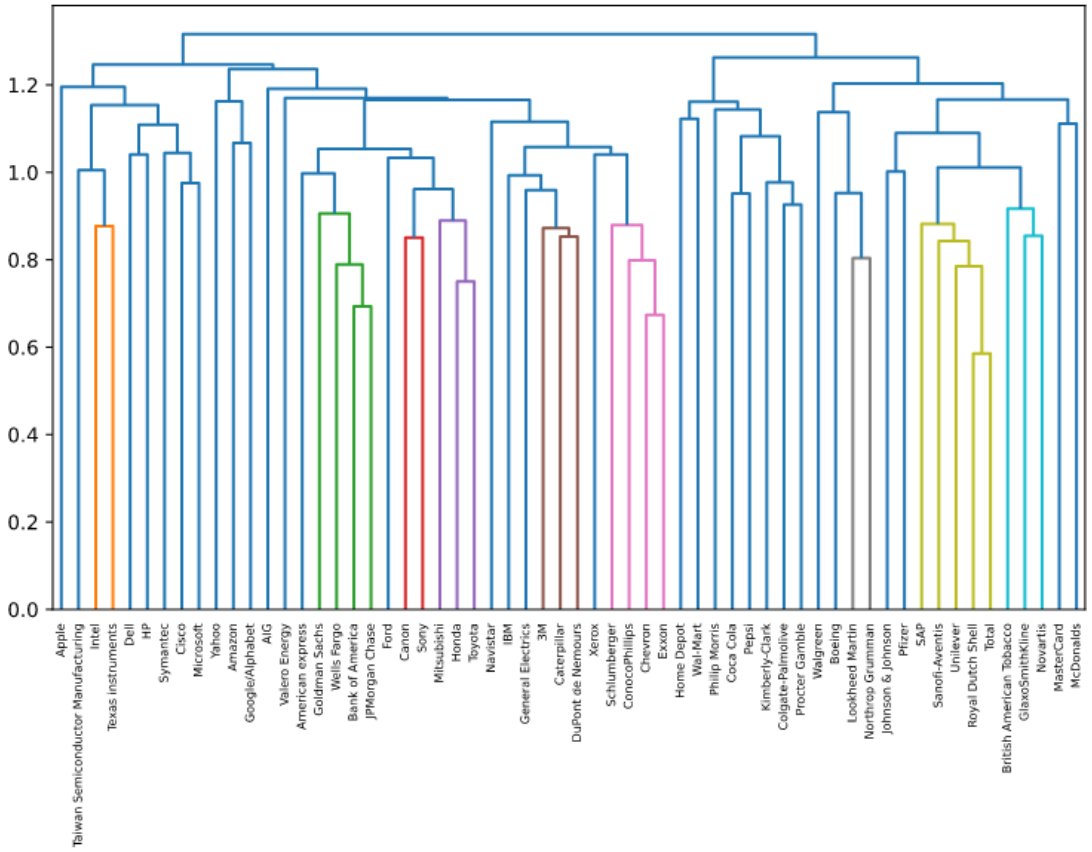
We use scipy in this example



Output dendrogram (agglomerative hierarchical clustering)

```
# Import normalize
from sklearn.preprocessing import normalize
# Normalize the movements: normalized_movements
normalized_movements = normalize(movements)
# Calculate the linkage: mergings
mergings = linkage(normalized_movements,
method='complete')
# Plot the dendrogram
dendrogram(mergings,
           labels=companies,
           leaf_rotation=90,
           leaf_font_size=6)
plt.show()
```

Applying normalization before hierarchical clustering



Output

Cluster labels in hierarchical clustering

In **complete linkage**, the distance between clusters is the distance between the **furthest** points of the clusters. In **single linkage**, the distance between clusters is the distance between the **closest** points of the clusters.

fcluster() function to extract the cluster labels for this intermediate clustering

```
# Perform the necessary imports
import pandas as pd
from scipy.cluster.hierarchy import fcluster
# Use fcluster to extract labels: labels
labels = fcluster(mergings, 6, criterion='distance')
# Create a DataFrame with labels and varieties as columns: df
df = pd.DataFrame({'labels': labels, 'varieties': varieties})
# Create crosstab: ct
ct = pd.crosstab(df['labels'], df['varieties'])
# Display ct
print(ct)
```

select “height” = 6

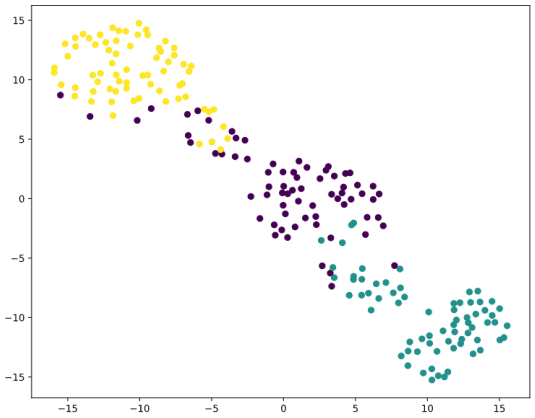
t-sne (t-distributed stochastic neighbor embedding)
https://bit.ly/2RvX5U6

- map samples to 2D space (or 3D)
- good for inspecting datasets
- ex. iris dataset -> 4 measurements (4 dimension) -> t-SNE maps to 2D space
- t-sne in sklearn has only fit_transform()
- need learning rate
- output different every time

```
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
model = TSNE(learning_rate=100)
transformed = model.fit_transform(samples)
xs = transformed[:,0]
ys = transformed[:,1]
plt.scatter(xs, ys, c=species)
plt.show()
```

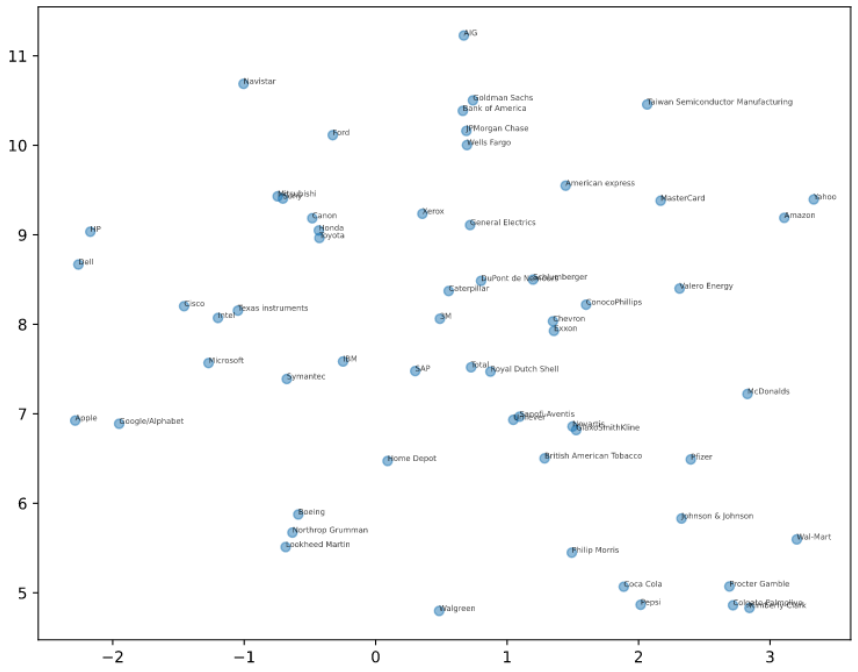
```
# Import TSNE
from sklearn.manifold import TSNE
# Create a TSNE instance: model
model = TSNE(learning_rate=200)
# Apply fit_transform to samples: tsne_features
tsne_features = model.fit_transform(samples)
# Select the 0th feature: xs
xs = tsne_features[:,0]
# Select the 1st feature: ys
ys = tsne_features[:,1]
# Scatter plot, coloring by variety_numbers
plt.scatter(xs,ys,c=variety_numbers)
plt.show()
```

Example of t-SNE



Output (will not be the same if run again)

```
# Import TSNE
from sklearn.manifold import TSNE
# Create a TSNE instance: model
model = TSNE(learning_rate=50)
# Apply fit_transform to normalized_movements:
tsne_features
tsne_features = model.fit_transform
(normalized_movements)
# Select the 0th feature: xs
xs = tsne_features[:,0]
# Select the 1th feature: ys
ys = tsne_features[:,1]
# Scatter plot
plt.scatter(xs,ys,alpha=0.5)
# Annotate the points
for x, y, company in zip(xs, ys, companies):
    plt.annotate(company, (x, y), fontsize=5,
alpha=0.75)
plt.show()
```



Dimension Reduction

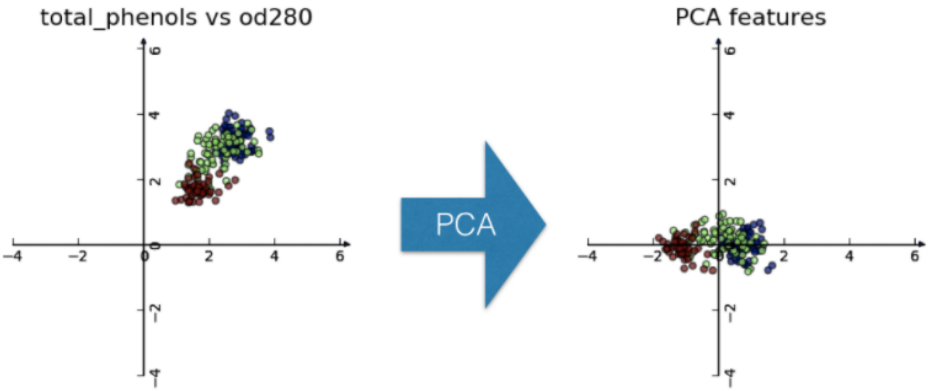
- remove noise
- more efficient storage and computation

Principal Component Analysis (PCA)

- fundamental dimension reduction technique
- first: decorrelation
- second: reduce dimensions

PCA align data with axes

- rotates data samples to be aligned with axes
- shifts data -> mean = 0
- no information lost

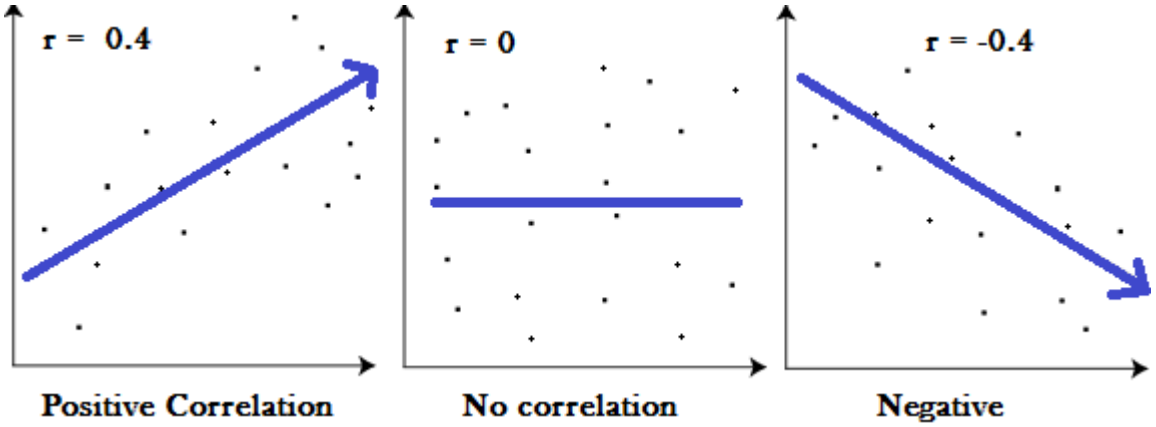


Decorrelation using PCA

“Principle Component” = direction of variance

Pearson correlation

- value between -1 to 1
- Correlation between sets of data is a measure of how well they are related. The most common measure of correlation in stats is the Pearson Correlation. The full name is the **Pearson Product Moment Correlation (PPMC)**. It shows the **linear relationship between two sets of data**.

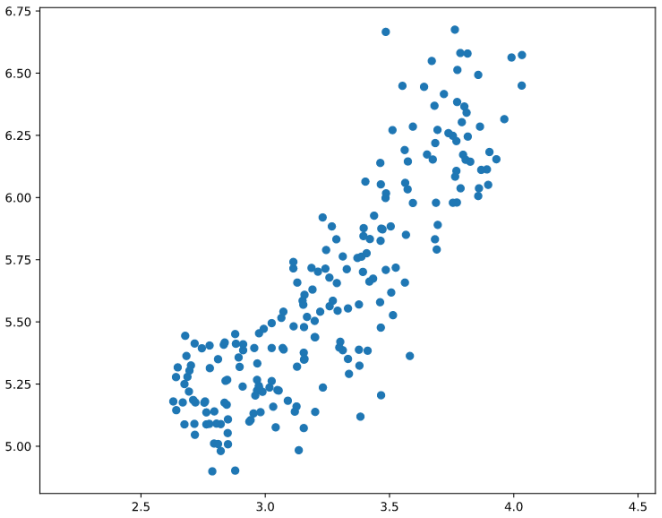


$$r = \frac{\sum (x - m_x)(y - m_y)}{\sqrt{\sum (x - m_x)^2 \sum (y - m_y)^2}}$$

x and y are two vectors of length n m, x and m, y corresponds to the means of x and y, respectively.

```
# Perform the necessary imports
import matplotlib.pyplot as plt
from scipy.stats import pearsonr
# Assign the 0th column of grains: width
width = grains[:,0]
# Assign the 1st column of grains: length
length = grains[:,1]
# Scatter plot width vs length
plt.scatter(width, length)
plt.axis('equal')
plt.show()
# Calculate the Pearson correlation
correlation, pvalue = pearsonr(width,length)
# Display the correlation
print(correlation)
```

Plotting the graph and calculating correlation value using pearsonr()



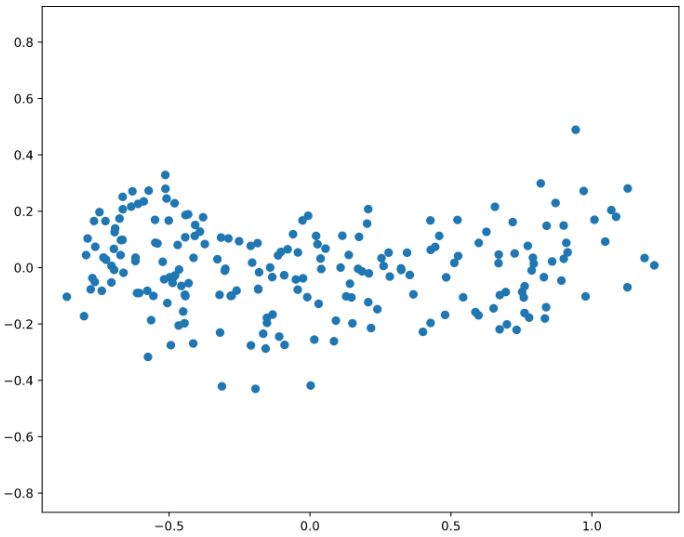
Output graph, we can see that it's clearly has the positive correlation

<script.py> output:
0.8604149377143466

value of correlation confirmed the “positive correlation”

```
# Import PCA
from sklearn.decomposition import PCA
# Create PCA instance: model
model = PCA()
# Apply the fit_transform method of model to
grains: pca_features
pca_features = model.fit_transform(grains)
# Assign 0th column of pca_features: xs
xs = pca_features[:,0]
# Assign 1st column of pca_features: ys
ys = pca_features[:,1]
# Scatter plot xs vs ys
plt.scatter(xs, ys)
plt.axis('equal')
plt.show()
# Calculate the Pearson correlation of xs and ys
correlation, pvalue = pearsonr(xs, ys)
# Display the correlation
print(correlation)
```

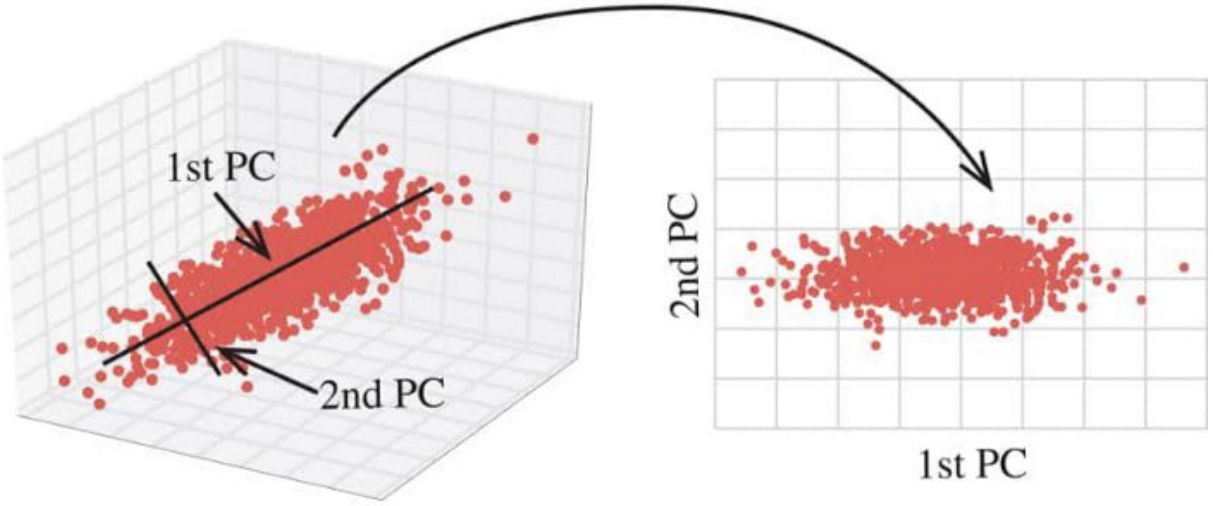
Applying PCA on grains data and using pearsonr() to calculate the correlation



Output graph (no correlation?)

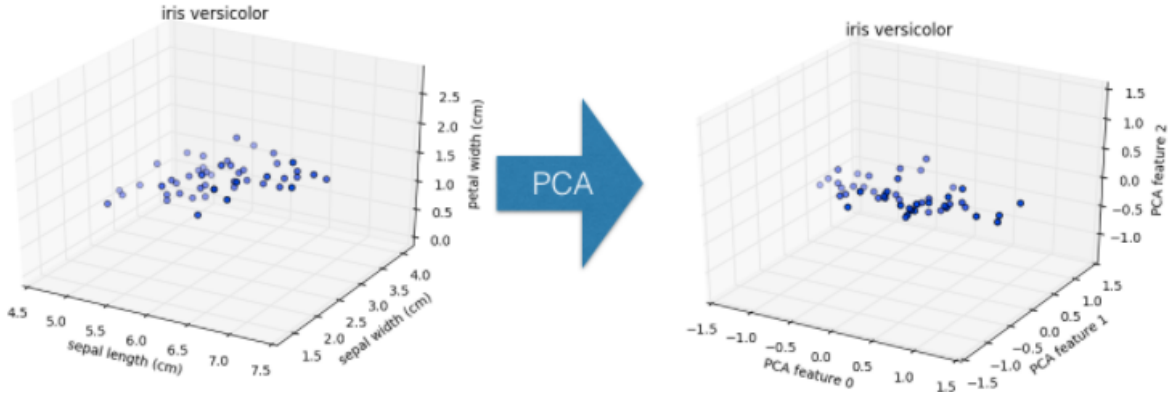
<script.py> output:
2.5478751053409354e-17

the correlation is positive (but very small value), so we can assume that there is no correlation??

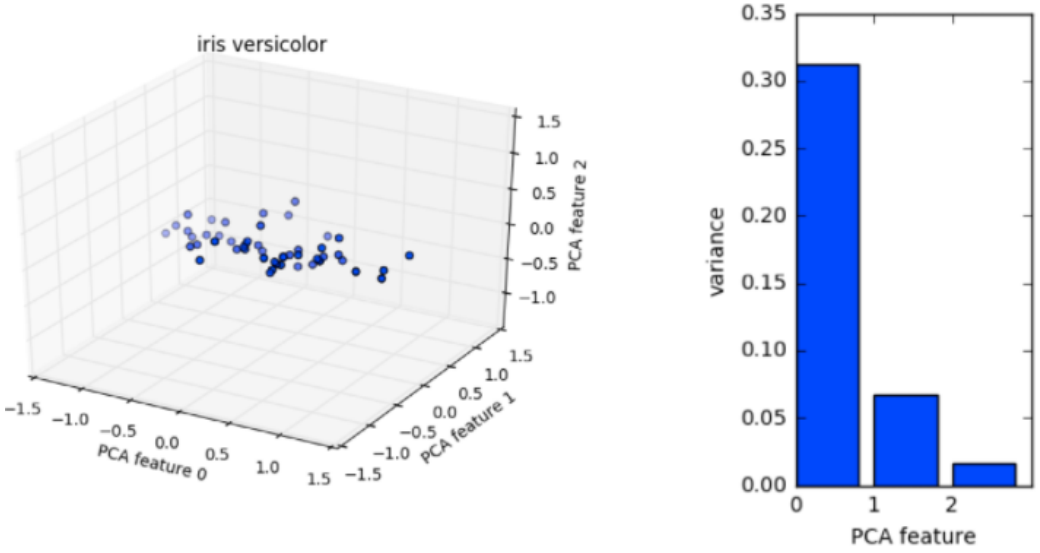


Intrinsic dimension

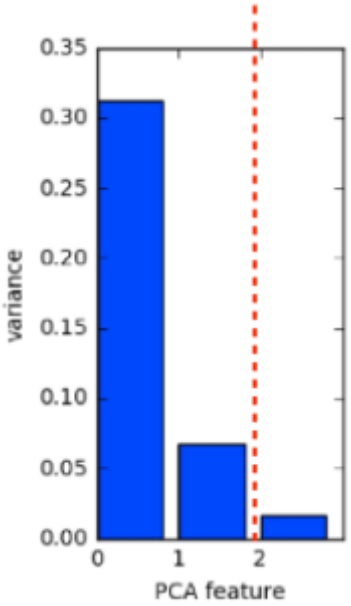
- number of features needed to approximate the dataset
- ex. lat and long -> displacement
- essential idea behind dimension reduction



PCA of versicolor example (data feature -> PCA feature)
PCA features are ordered by **variance descending**



Intrinsic dimension is number of PCA features with **significant variance**

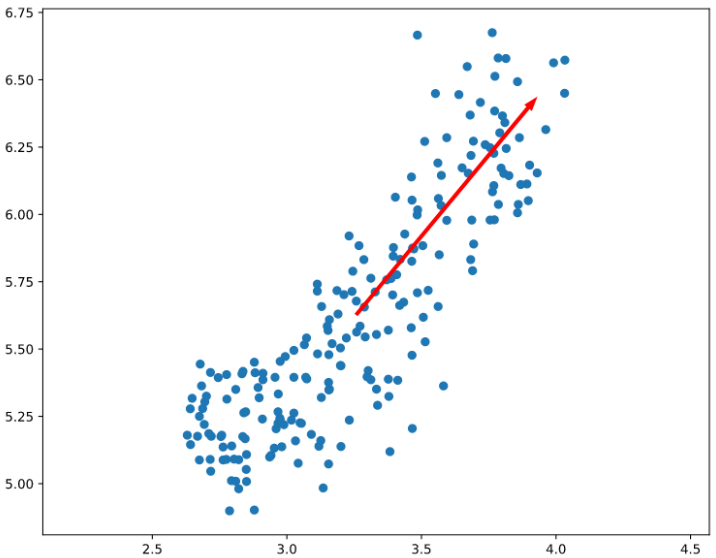


Here, first two PCA is significant -> intrinsic dimension is 2

- intrinsic dimension is an idealization
- answer not always one correct answer, maybe 2, 3, ...

Plotting “first PCA arrow”

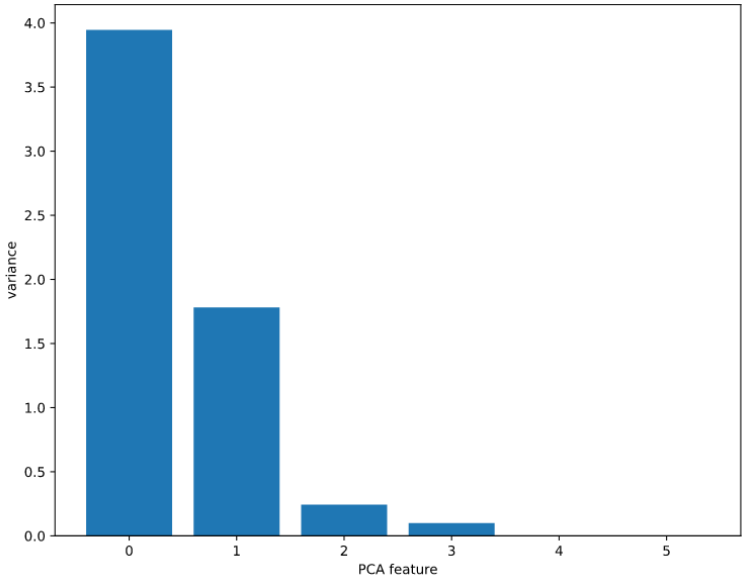
```
# Make a scatter plot of the untransformed points
plt.scatter(grains[:,0], grains[:,1])
# Create a PCA instance: model
model = PCA()
# Fit model to points
model.fit(grains)
# Get the mean of the grain samples: mean
mean = model.mean_
# Get the first principal component: first_pc
first_pc = model.components_[0,:]
# Plot first_pc as an arrow, starting at mean
plt.arrow(mean[0], mean[1], first_pc[0], first_pc
[1], color='red', width=0.01)
# Keep axes on same scale
plt.axis('equal')
plt.show()
```



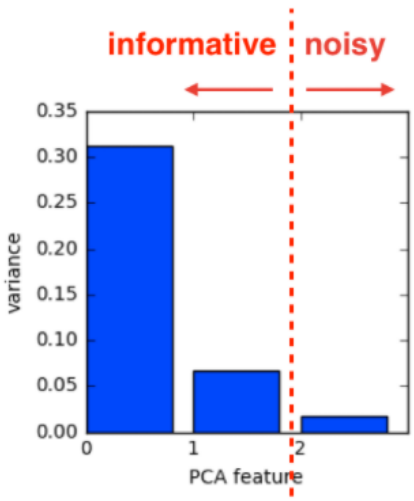
Variance of PCA features

```
# Perform the necessary imports
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
import matplotlib.pyplot as plt
# Create scaler: scaler
scaler = StandardScaler()
# Create a PCA instance: pca
pca = PCA()
# Create pipeline: pipeline
pipeline = make_pipeline(scaler,pca)
# Fit the pipeline to 'samples'
pipeline.fit(samples)
# Plot the explained variances
features = range(pca.n_components_)
plt.bar(features, pca.explained_variance_)
plt.xlabel('PCA feature')
plt.ylabel('variance')
plt.xticks(features)
plt.show()
```

Scale and PCA in the pipeline -> fit, then plot the “explained variance (y-axis)” by range of number of pca components (x-axis)



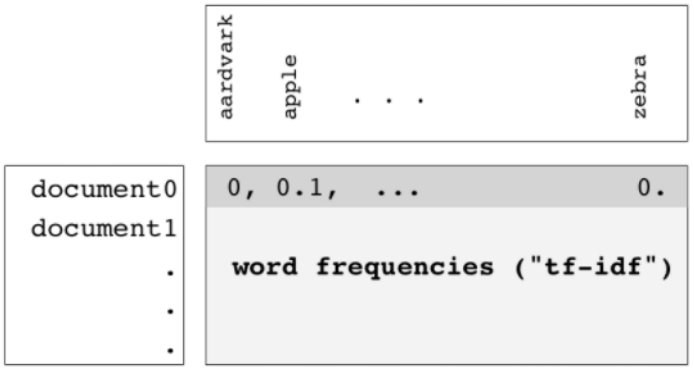
Dimension reduction -> intrintsic dimension = informative, other = noisy (assume)



PCA(n_componets=2) -> keep 2 features

Word frequency array

- rows = document, columns = words
- entries measure presence of each word in each document
- measure using “tf-idf”



- “Sparse” = most entries are zero
- **scipy.sparse.csr_matrix** instead of numpy array
- csr_matrix remember only the non-zero entries (save space)

TruncatedSVD and csr_matrix

- scikit-learn `PCA` doesn't support `csr_matrix`
- Use scikit-learn `TruncatedSVD` instead
- Performs same transformation

```
from sklearn.decomposition import TruncatedSVD
model = TruncatedSVD(n_components=3)
model.fit(documents)  # documents is csr_matrix
TruncatedSVD(algorithm='randomized', ... )
transformed = model.transform(documents)
```

```
# Import PCA
from sklearn.decomposition import PCA
# Create a PCA model with 2 components: pca
pca = PCA(n_components=2)
# Fit the PCA instance to the scaled samples
pca.fit(scaled_samples)
# Transform the scaled samples: pca_features
pca_features = pca.transform(scaled_samples)
# Print the shape of pca_features
print(pca_features.shape)
```

Example of dimension reduction

```
# Import TfidfVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
# Create a TfidfVectorizer: tfidf
tfidf = TfidfVectorizer()
# Apply fit_transform to document: csr_mat
csr_mat = tfidf.fit_transform(documents)
# Print result of toarray() method
print(csr_mat.toarray())
# Get the words: words
words = tfidf.get_feature_names()
# Print words
print(words)
```

tf-idf word frequency

```
['cats say meow', 'dogs say woof', 'dogs chase cats']

<script.py> output:
    [[0.51785612 0.         0.         0.68091856 0.51785612 0.        ]
     [0.         0.         0.51785612 0.         0.51785612 0.68091856]
     [0.51785612 0.68091856 0.51785612 0.         0.         0.        ]]
    ['cats', 'chase', 'dogs', 'meow', 'say', 'woof']
```

```python
# Perform the necessary imports
from sklearn.decomposition import TruncatedSVD
from sklearn.cluster import KMeans
from sklearn.pipeline import make_pipeline
# Create a TruncatedSVD instance: svd
svd = TruncatedSVD(n_components=50)
# Create a KMeans instance: kmeans
kmeans = KMeans(n_clusters=6)
# Create a pipeline: pipeline
pipeline = make_pipeline(svd, kmeans)
```

TruncatedSVD is able to perform PCA on sparse arrays in csr_matrix format, such as word-frequency arrays.

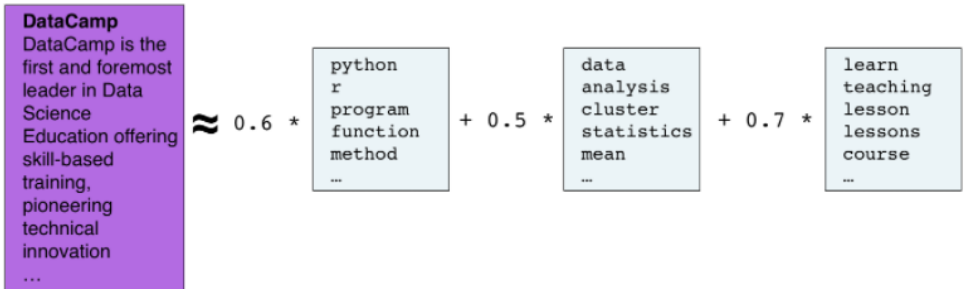
```python
# Import pandas
import pandas as pd
# Fit the pipeline to articles
pipeline.fit(articles)
# Calculate the cluster labels: labels
labels = pipeline.predict(articles)
# Create a DataFrame aligning labels and titles: df
df = pd.DataFrame({'label': labels, 'article': titles})
# Display df sorted by cluster label
print(df.sort_values('label'))
```

Fit the pipeline, predict to get labels, create DataFrame, print out the df sorted by ‘label’

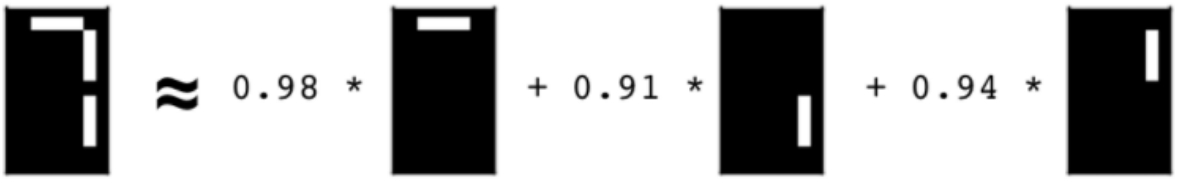
Non-negative matrix factorization (NMF)

- dimension reduction technique
- interpretable (unlike PCA) (and easy to interpret)
- !! all sample features must be non-negative (>=0)

NMF expresses documents as combinations of topics “themes”

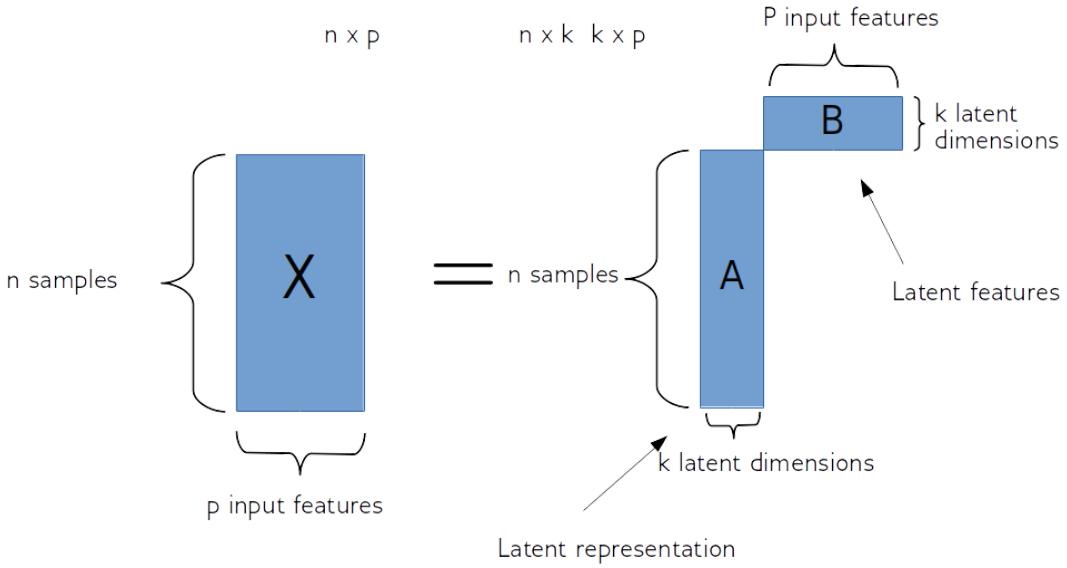


NMF expresses images as combinations of patterns



$$X = AB$$

n x p n x k k x p



scikitlearn NMF

- specify n_components
- works with numpy arrays and csr_matrix

```
# Import NMF
from sklearn.decomposition import NMF
# Create an NMF instance: model
model = NMF(n_components=6)
# Fit the model to articles
model.fit(articles)
# Transform the articles: nmf_features
nmf_features = model.transform(articles)
# Print the NMF features
print(nmf_features.round(2))
```

```
# Import pandas
import pandas as pd
# Create a pandas DataFrame: df
df = pd.DataFrame(nmf_features,index=titles)
# Print the row for 'Anne Hathaway'
print(df.loc['Anne Hathaway'])
# Print the row for 'Denzel Washington'
print(df.loc['Denzel Washington'])
```

<script.py> output:

```
0    0.003845
1    0.000000
2    0.000000
3    0.575711
4    0.000000
5    0.000000
Name: Anne Hathaway, dtype: float64
0    0.000000
1    0.005601
2    0.000000
3    0.422380
4    0.000000
```

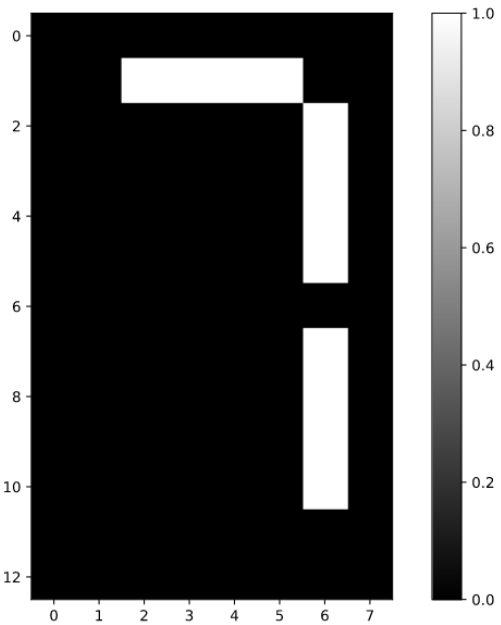
```
5    0.000000
Name: Denzel Washington, dtype: float64
```

Notice that for both actors, the NMF feature 3 has by far the highest value. This means that both articles are reconstructed using mainly the 3rd NMF component.

```
# Import pandas
import pandas as pd
# Create a DataFrame: components_df
components_df = pd.DataFrame(model.components_, columns=words)
# Print the shape of the DataFrame
print(components_df.shape)
# Select row 3: component
component = components_df.iloc[3,:]
# Print result of nlargest
print(component.nlargest())
```

```
# Import pyplot
from matplotlib import pyplot as plt
# Select the 0th row: digit
digit = samples[0,:]
# Print digit
print(digit)
# Reshape digit to a 13x8 array: bitmap
bitmap = digit.reshape(13, 8)
# Print bitmap
print(bitmap)
# Use plt.imshow to display bitmap
plt.imshow(bitmap, cmap='gray',
interpolation='nearest')
plt.colorbar()
plt.show()
```

Select “digit” row from digit table (select row 0) -> reshape to 13x8 -> print

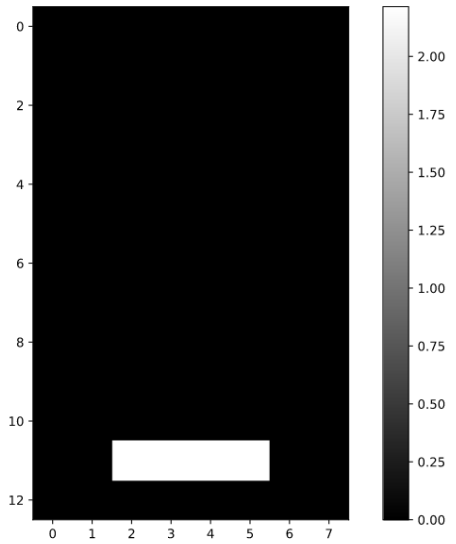


Output has row = 13 (0-12), col = 8 (0-7)

NMF learn part of an image

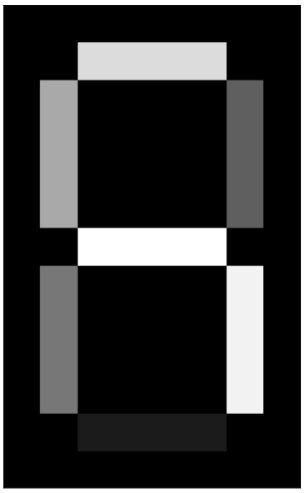
```
def show_as_image(sample):
    bitmap = sample.reshape((13, 8))
    plt.figure()
    plt.imshow(bitmap, cmap='gray', interpolation='nearest')
    plt.colorbar()
    plt.show()
```

```
# Import NMF
from sklearn.decomposition import NMF
# Create an NMF model: model
model = NMF(n_components=7)
# Apply fit_transform to samples: features
features = model.fit_transform(samples)
# Call show_as_image on each component
for component in model.components_:
    show_as_image(component)
# Assign the 0th row of features: digit_features
digit_features = features[0,:]
# Print digit_features
print(digit_features)
```



PCA doesn't learn parts

```
# Import PCA
from sklearn.decomposition import PCA
# Create a PCA instance: model
model = PCA(n_components=7)
# Apply fit_transform to samples: features
features = model.fit_transform(samples)
# Call show_as_image on each component
for component in model.components_:
    show_as_image(component)
```



Notice that the components of PCA do not represent meaningful parts of images of LED digits

Building recommender systems using NMF

- task: recommend articles similar to the article being read by customer
- strategy: apply NMF to the word-frequency array

Version of articles

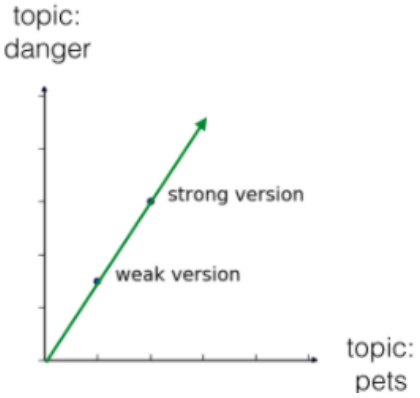
- one version might uses many meaningless words
- BUT all version lie on the same line through the origin

strong version

Dog bites man! Attack by terrible canine leaves man paralyzed...

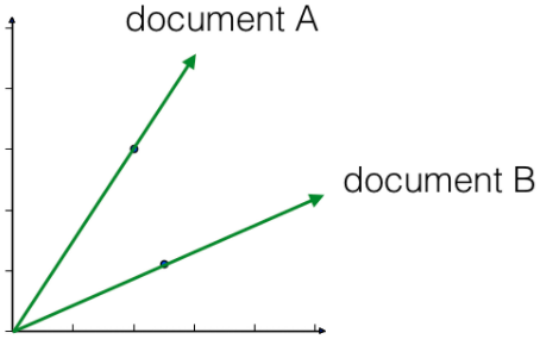
weak version

You may have heard, unfortunately it seems that a dog has perhaps bitten a man ...



Cosine similarity

- uses the angle between the lines
- higher value = more similar
- max = 1, when angle is 0 degree



Which articles are similar to ‘Cristiano Ronaldo’?

```
# Perform the necessary imports
import pandas as pd
from sklearn.preprocessing import normalize
# Normalize the NMF features: norm_features
norm_features = normalize(nmf_features)
# Create a DataFrame: df
df = pd.DataFrame(norm_features, index=titles)
# Select the row corresponding to 'Cristiano Ronaldo': article
article = df.loc['Cristiano Ronaldo']
# Compute the dot products: similarities
similarities = df.dot(article)
# Display those with the largest cosine similarity
print(similarities.nlargest())
```

normalize NMF features -> create DataFrame -> apply dot product

<script.py> output:

```
Cristiano Ronaldo             1.000000
Franck Ribéry                 0.999972
Radamel Falcao                0.999942
Zlatan Ibrahimović            0.999942
France national football team 0.999923
dtype: float64
```

Recommend music artist

```
# Perform the necessary imports
from sklearn.decomposition import NMF
from sklearn.preprocessing import Normalizer, MaxAbsScaler
from sklearn.pipeline import make_pipeline
# Create a MaxAbsScaler: scaler
scaler = MaxAbsScaler()
# Create an NMF model: nmf
nmf = NMF(n_components=20)
# Create a Normalizer: normalizer
normalizer = Normalizer()
# Create a pipeline: pipeline
pipeline = make_pipeline(scaler, nmf, normalizer)
# Apply fit_transform to artists: norm_features
norm_features = pipeline.fit_transform(artists)
```

```
# Import pandas
import pandas as pd
# Create a DataFrame: df
df = pd.DataFrame(norm_features, index=artist_names)
# Select row of 'Bruce Springsteen': artist
artist = df.loc['Bruce Springsteen']
# Compute cosine similarities: similarities
similarities = df.dot(artist)
# Display those with highest cosine similarity
print(similarities.nlargest())
```

<script.py> output:

```
Bruce Springsteen    1.000000
Neil Young           0.955896
Van Morrison         0.872452
```

```
Leonard Cohen       0.864763
Bob Dylan           0.859047
dtype: float64
```

Linear Classifier in Python

LogisticRegression example

```
import sklearn.datasets
wine = sklearn.datasets.load_wine()
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(wine.data, wine.target)
lr.score(wine.data, wine.target)
```

```
0.972
```

```
lr.predict_proba(wine.data[:1])
```

```
array([[  9.951e-01,   4.357e-03,   5.339e-04]])
```

`LinearSVC` works the same way:

```
import sklearn.datasets

wine = sklearn.datasets.load_wine()
from sklearn.svm import LinearSVC

svm = LinearSVC()

svm.fit(wine.data, wine.target)
svm.score(wine.data, wine.target)
```

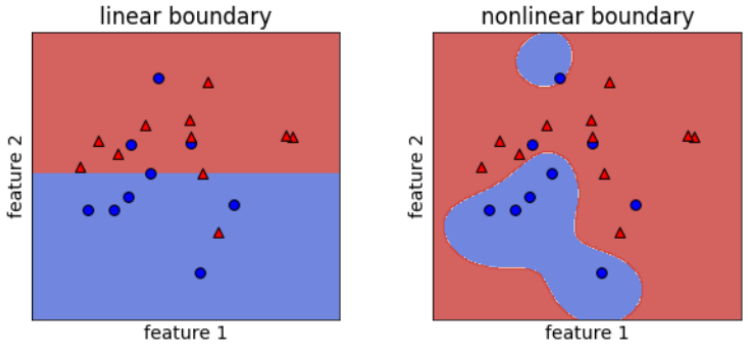
```
0.893
```

```
import sklearn.datasets
wine = sklearn.datasets.load_wine()
from sklearn.svm import SVC
svm = SVC() # default hyperparameters
svm.fit(wine.data, wine.target);
svm.score(wine.data, wine.target)
```

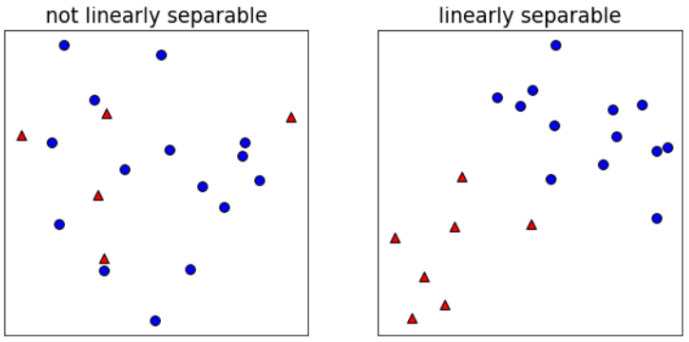
```
1.
```

```
from sklearn import datasets
digits = datasets.load_digits()
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target)
# Apply logistic regression and print scores
lr = LogisticRegression()
lr.fit(X_train, y_train)
print(lr.score(X_train, y_train))
print(lr.score(X_test, y_test))
# Apply SVM and print scores
svm = SVC()
svm.fit(X_train, y_train)
print(svm.score(X_train, y_train))
print(svm.score(X_test, y_test))
```

Linear decision boundaries



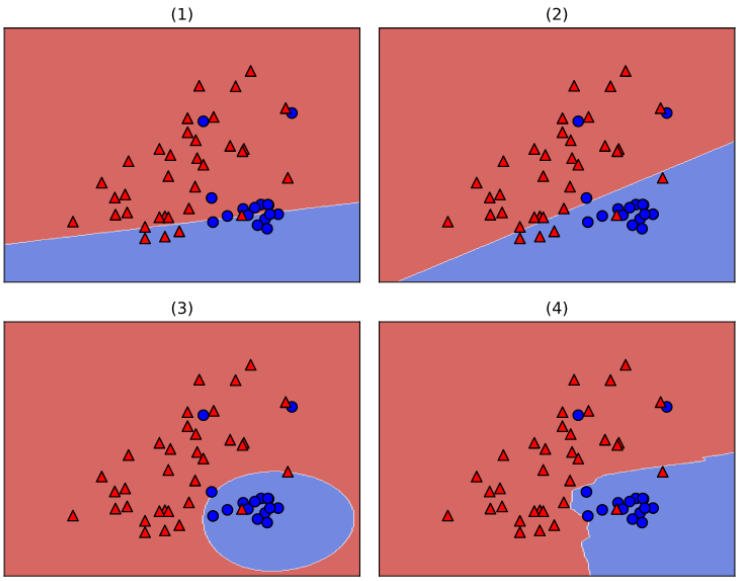
Linearly separable data



Comparing decision boundaries between each classifiers

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC, LinearSVC
from sklearn.neighbors import KNeighborsClassifier

# Define the classifiers
classifiers = [LogisticRegression(), LinearSVC(), SVC(),
KNeighborsClassifier()]
# Fit the classifiers
for c in classifiers:
    c.fit(X, y)
# Plot the classifiers
plot_4_classifiers(X, y, classifiers)
plt.show()
```

We got 4 classifiers, then call each using for loop



4 classifiers boundary, respectively

Linear classifier prediction

- raw model output = coef . features + intercept
- check raw model output “sign”
- positive: predict one class
- negative: predict the other class
- same for LogReg and LinearSVM (fit diff, predict same)
- raw output = 0? it's PERFECTLY at the prediction line

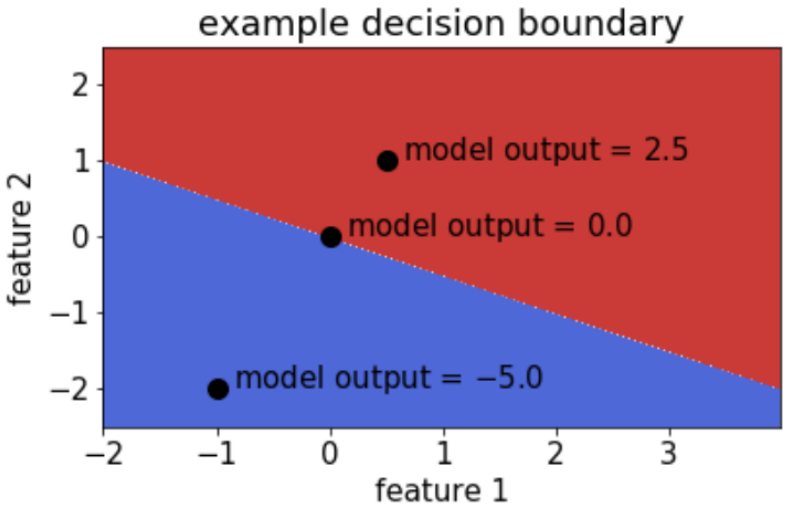
```
lr.coef_ @ X[10] + lr.intercept_ # raw model output
```

```
array([-33.78572166])
```

```
lr.coef_ @ X[20] + lr.intercept_ # raw model output
```

```
array([ 0.08050621])
```

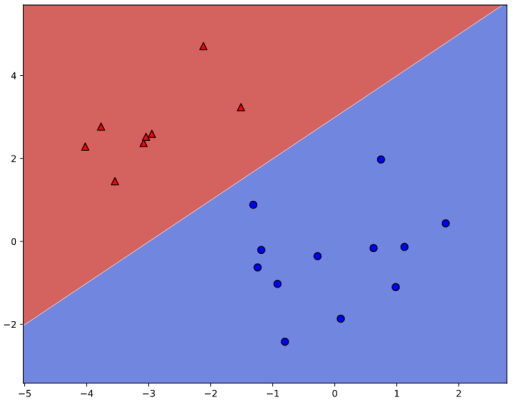
Pos and neg values are present



Pos = THIS class, Neg = other class, 0 = exactly at the prediction line

```
# Set the coefficients
model.coef_ = np.array([[-1,1]])
model.intercept_ = np.array([-3])
# Plot the data and decision boundary
plot_classifier(X,y,model)
# Print the number of errors
num_err = np.sum(y != model.predict(X))
print("Number of errors:", num_err)
```

Manually set the coef_ and intercept_ to match the scatter plots



Output boundary

Loss function

Least square: the squared loss

- LogReg minimize loss

$$\sum_{i=1}^{n}(\text{true } i\text{th target value} - \text{predicted } i\text{th target value})^2$$

-
- Minimization with respect to coef or parameters of the model
- sklearn “model.score()” isn’t a loss function

Classification: The 0-1 loss

- squared loss is not appropriate for classification
- natural loss = number of errors
- 0-1 loss: 0 = correct prediction, 1 = incorrect prediction
- loss is hard to minimize

```
from scipy.optimize import minimize
```

```
minimize(np.square, 0).x
```

```
array([0.])
```

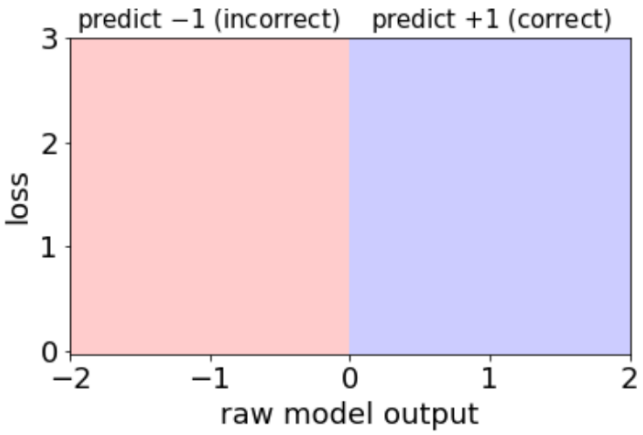
```
minimize(np.square, 2).x
```

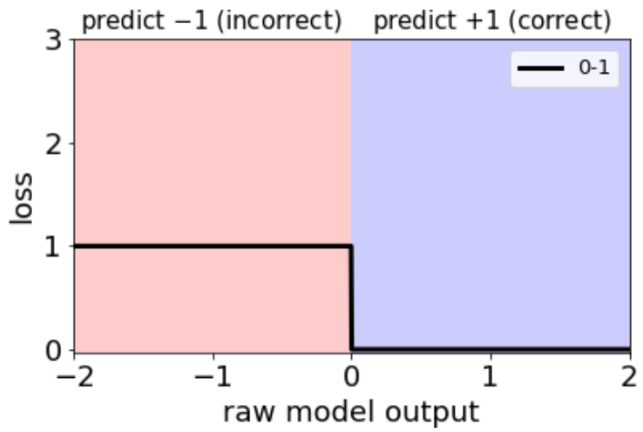
```
array([-1.88846401e-08])
```

```python
# The squared error, summed over training examples
def my_loss(w):
    s = 0
    for i in range(y.size):
        # Get the true and predicted target values for example 'i'
        y_i_true = y[i]
        y_i_pred = w@X[i]
        s = s + (y_i_pred-y_i_true)**2
    return s
# Returns the w that makes my_loss(w) smallest
w_fit = minimize(my_loss, X[0]).x
print(w_fit)
# Compare with scikit-learn's LinearRegression coefficients
lr = LinearRegression(fit_intercept=False).fit(X,y)
print(lr)
```

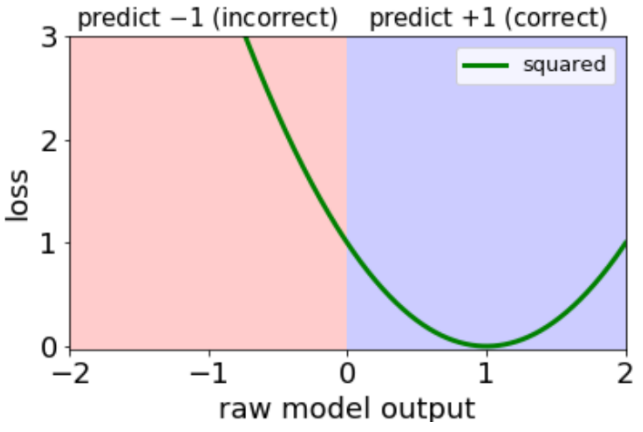
Squared error, then compare minimizing loss with minimize() and LogReg()

The raw model output

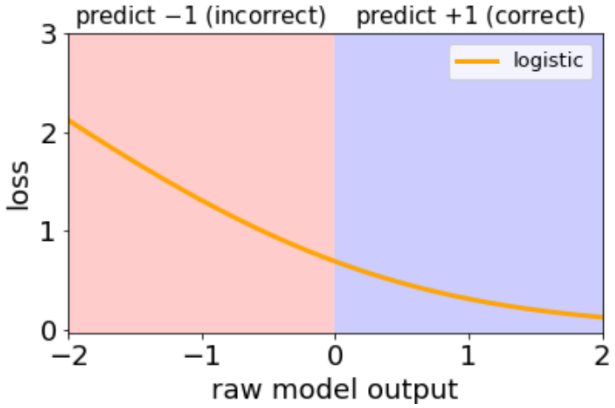




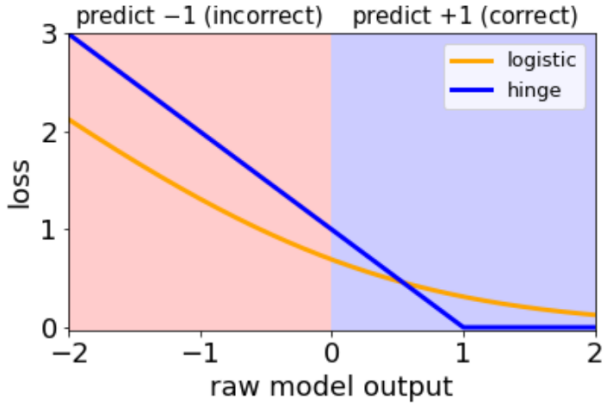
0-1 loss diagram



linear regression loss diagram



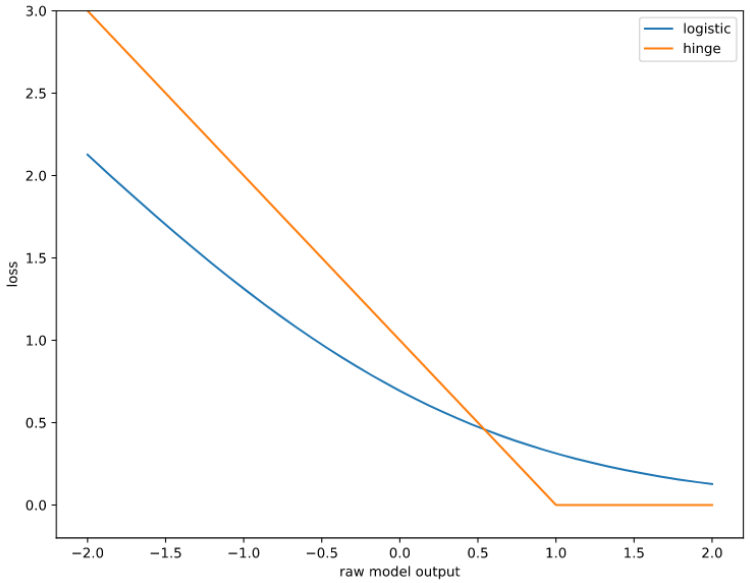
logistic loss diagram



hinge loss diagram comparing to logistic

```
# Mathematical functions for logistic and hinge losses
def log_loss(raw_model_output):
    return np.log(1+np.exp(-raw_model_output))
def hinge_loss(raw_model_output):
    return np.maximum(0,1-raw_model_output)

# Create a grid of values and plot
grid = np.linspace(-2,2,1000)
plt.plot(grid, log_loss(grid), label='logistic')
plt.plot(grid, hinge_loss(grid), label='hinge')
plt.legend()
plt.show()
```

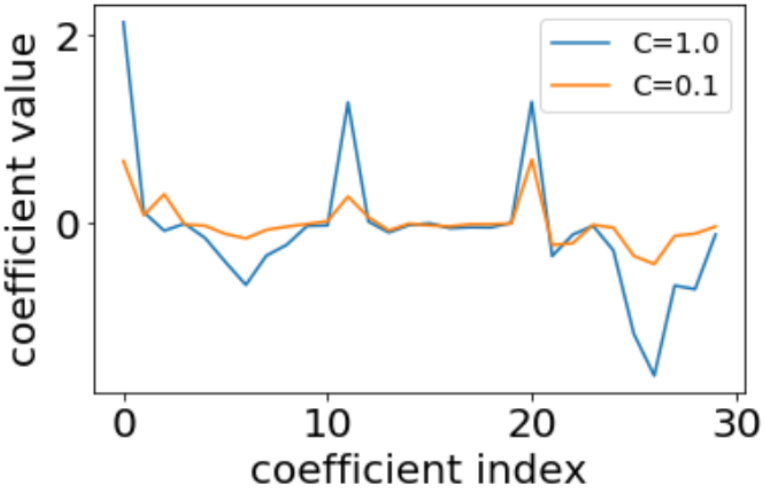
Comparing logistic and hinge loss



```python
# The logistic loss, summed over training examples
def my_loss(w):
    s = 0
    for i in range(y.size):
        raw_model_output = w@X[i]
        s = s + log_loss(raw_model_output * y[i])
    return s
# Returns the w that makes my_loss(w) smallest
w_fit = minimize(my_loss, X[0]).x
print(w_fit)
# Compare with scikit-learn's LogisticRegression
lr = LogisticRegression(fit_intercept=False, C=1000000).fit(X,y)
print(lr.coef_)
```

Implementing logistic regression -> logistic loss

Regularized logistic regression



How does regularization affect training accuracy?

Training accuracy

```python
lr_weak_reg = LogisticRegression(C=100)
lr_strong_reg = LogisticRegression(C=0.01)

lr_weak_reg.fit(X_train, y_train)
lr_strong_reg.fit(X_train, y_train)

lr_weak_reg.score(X_train, y_train)
lr_strong_reg.score(X_train, y_train)
```

```
1.0
0.92
```

$$\text{regularized loss} = \text{original loss} + \text{large coefficient penalty}$$

- more regularization: lower training accuracy

Testing accuracy

```
lr_weak_reg.score(X_test, y_test)
```

```
0.86
```

```
lr_strong_reg.score(X_test, y_test)
```

```
0.88
```

$$\text{regularized loss} = \text{original loss} + \text{large coefficient penalty}$$

- more regularization: lower training accuracy
- more regularization: (almost always) higher test accuracy

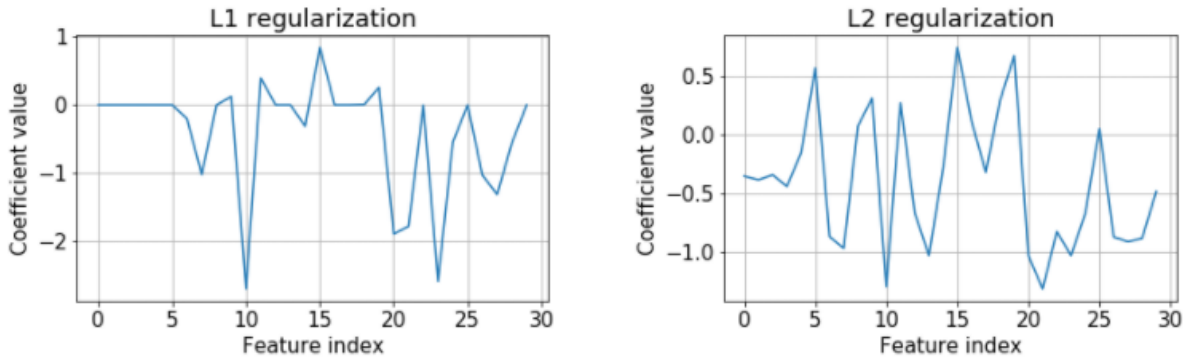
L1 vs L2 regularization (Lasso vs Ridge)

- Lasso = linear regression with L1 regularization
- Ridge = linear regression with L2 regularization
- For other models like logistic regression we just say L1, L2, etc.

```
lr_L1 = LogisticRegression(penalty='l1')
lr_L2 = LogisticRegression() # penalty='l2' by default

lr_L1.fit(X_train, y_train)
lr_L2.fit(X_train, y_train)
```

```
plt.plot(lr_L1.coef_.flatten())
plt.plot(lr_L2.coef_.flatten())
```

Lasso vs Ridge regularization

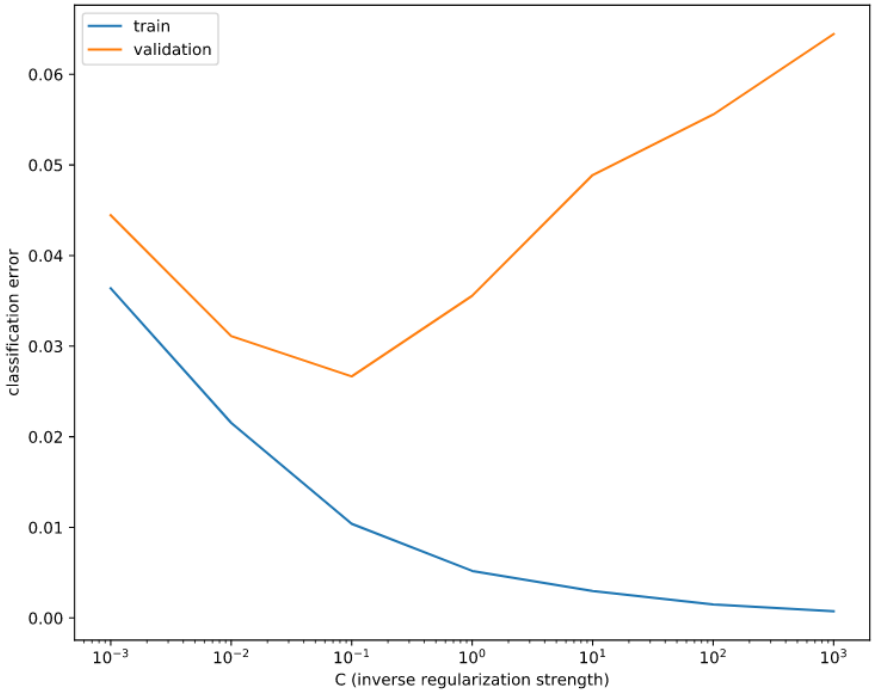


Graph of Lasso and Ridge regularization, respectively

Comparing deltas in regularized logistic regression

```
# Train and validaton errors initialized as empty list
train_errs = list()
valid_errs = list()
# Loop over values of C_value
for C_value in [0.001, 0.01, 0.1, 1, 10, 100, 1000]:
    # Create LogisticRegression object and fit
    lr = LogisticRegression(C=C_value)
    lr.fit(X_train, y_train)
    # Evaluate error rates and append to lists
    train_errs.append( 1.0 - lr.score(X_train, y_train) )
    valid_errs.append( 1.0 - lr.score(X_valid, y_valid) )
# Plot results
plt.semilogx(C_values, train_errs, C_values, valid_errs)
plt.legend(("train", "validation"))
plt.show()
```

Comparing values in C_value



Logistic regression and feature selection

```
# Specify L1 regularization
lr = LogisticRegression(penalty='l1')
# Instantiate the GridSearchCV object and run the search
searcher = GridSearchCV(lr, {'C':[0.001, 0.01, 0.1, 1, 10]})
searcher.fit(X_train, y_train)
# Report the best parameters
print("Best CV params", searcher.best_params_)
# Find the number of nonzero coefficients (selected features)
best_lr = searcher.best_estimator_
coefs = best_lr.coef_
print("Total number of features:", coefs.size)
print("Number of selected features:", np.count_nonzero(coefs))
```

Search for best value of C using Grid Search

<script.py> output:
Best CV params {'C': 1}
Total number of features: 2500
Number of selected features: 1220

Identifying the most positive and negative vocabs (words)

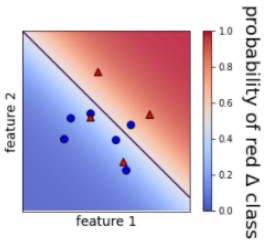
```
# Get the indices of the sorted cofficients
inds_ascending = np.argsort(lr.coef_.flatten())
inds_descending = inds_ascending[::-1]
# Print the most positive words
print("Most positive words: ", end="")
for i in range(5):
    print(vocab[inds_descending[i]], end=", ")
print("\n")
# Print most negative words
print("Most negative words: ", end="")
for i in range(5):
    print(vocab[inds_ascending[i]], end=", ")
print("\n")
```

<script.py> output:
Most positive words: favorite, superb, noir, knowing, loved,

Most negative words: disappointing, waste, worst, boring, lame,

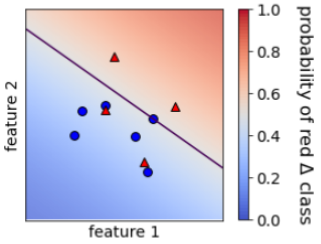
Logistic regression probabilities

Without regularization $(C = 10^8)$:



- model coefficients: `[[1.55 1.57]]`
- model intercept: `[-0.64]`

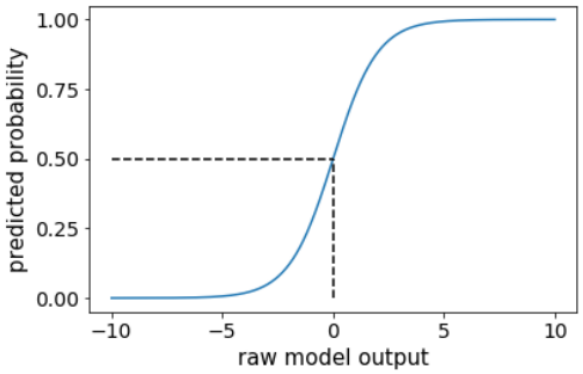
With regularization $(C = 1)$:



- model coefficients: `[[0.45 0.64]]`
- model intercept: `[-0.26]`

logistic regression predictions: sign of raw model output

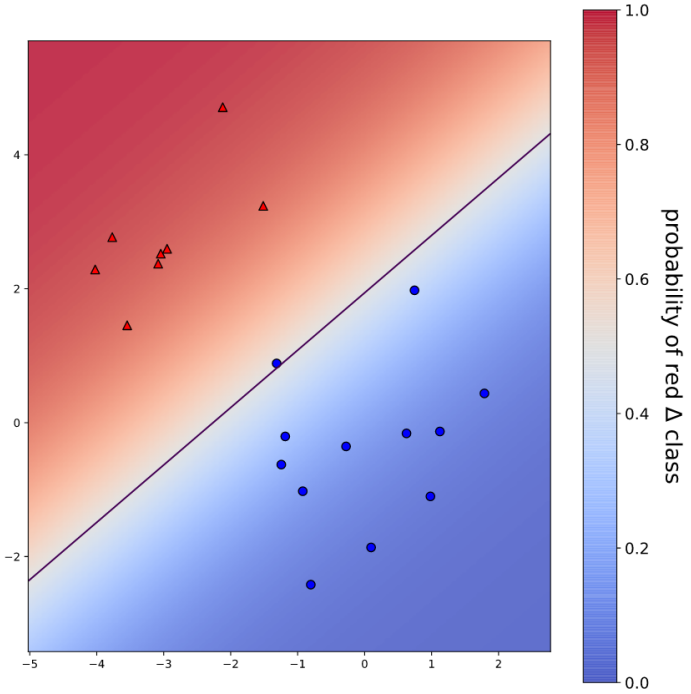
logistic regression probabilities: "squashed" raw model output



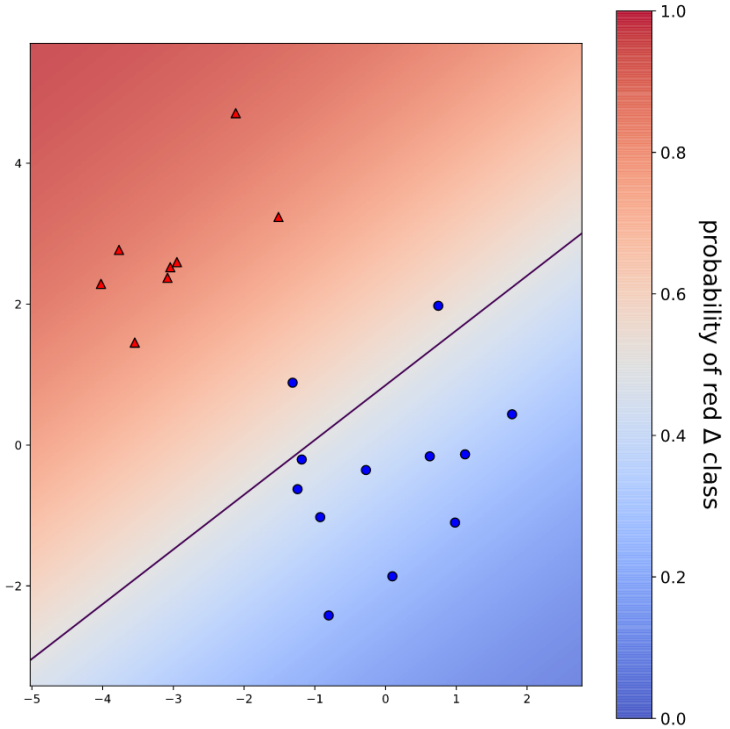
Comparing regularization strength

```
# Set the regularization strength
model = LogisticRegression(C=0.1)
# Fit and plot
model.fit(X,y)
plot_classifier(X,y,model,proba=True)
# Predict probabilities on training points
prob = model.predict_proba(X)
print("Maximum predicted probability", np.max(prob))
```

Compare between C=100, and C=0.1



C=100 plot

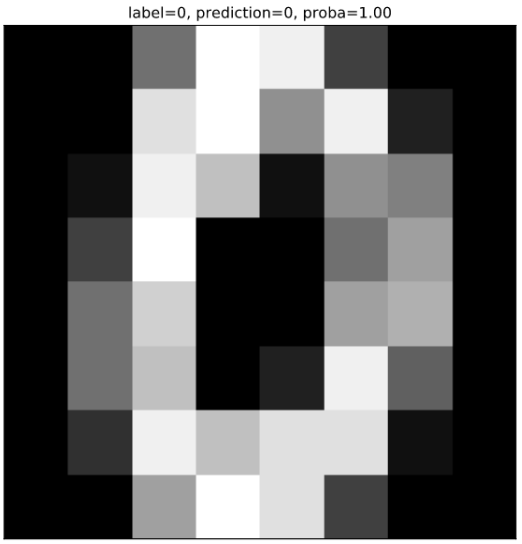


C=0.1 plot

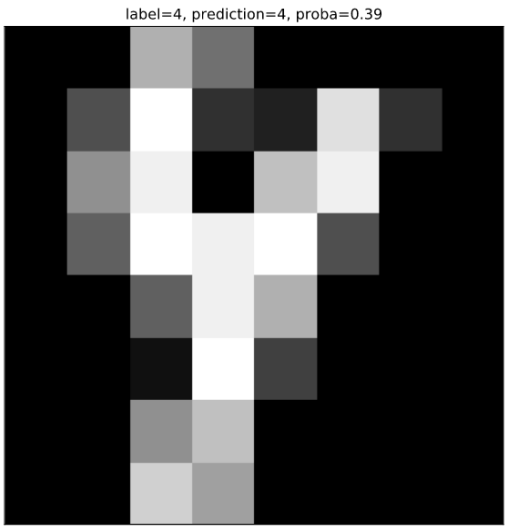
Visualizing easy and difficult examples

```
lr = LogisticRegression()
lr.fit(X,y)

# Get predicted probabilities
proba = lr.predict_proba(X)

# Sort the example indices by their maximum probability
proba_inds = np.argsort(np.max(proba,axis=1))

# Show the most confident (least ambiguous) digit
show_digit(proba_inds[-1], lr)

# Show the least confident (most ambiguous) digit
show_digit(proba_inds[0], lr)
```

Show most and least confident



Most confident (proba=1.00)



Least confident (proba=0.39)

Multi-class logistic regression
-> Combining binary classifiers with one-vs-rest

```
lr0.fit(X, y==0)

lr1.fit(X, y==1)

lr2.fit(X, y==2)
```

```
# get raw model output
lr0.decision_function(X)[0]
```

```
6.124
```

```
lr1.decision_function(X)[0]
```

```
-5.429
```

```
lr2.decision_function(X)[0]
```

```
-7.532
```

```
lr.fit(X, y)
lr.predict(X)[0]
```

```
0
```

One-vs-rest:

- fit a binary classifier for each class
- predict with all, take largest output
- pro: simple, modular
- con: not directly optimizing accuracy
- common for SVMs as well
- can produce probabilities

"Multinomial" or "softmax":

- fit a single classifier for all classes
- prediction directly outputs best class
- con: more complicated, new code
- pro: tackle the problem directly
- possible for SVMs, but less common
- can produce probabilities

Model coefficients for multi-class

```
# one-vs-rest by default
lr_ovr = LogisticRegression()

lr_ovr.fit(X,y)

lr_ovr.coef_.shape
```

```
(3,13)
```

```
lr_ovr.intercept_.shape
```

```
(3,)
```

```
lr_mn = LogisticRegression(
   multi_class="multinomial",
   solver="lbfgs")
lr_mn.fit(X,y)

lr_mn.coef_.shape
```

```
(3,13)
```

```
lr_mn.intercept_.shape
```

```
(3,)
```

Comparing one-vs-rest to softmax classifier

```
# Fit one-vs-rest logistic regression classifier
lr_ovr = LogisticRegression()
lr_ovr.fit(X_train, y_train)
print("OVR training accuracy:", lr_ovr.score(X_train, y_train))
print("OVR test accuracy    :", lr_ovr.score(X_test, y_test))
# Fit softmax classifier
lr_mn = LogisticRegression(multi_class='multinomial', solver='lbfgs')
lr_mn.fit(X_train, y_train)
print("Softmax training accuracy:", lr_mn.score(X_train, y_train))
print("Softmax test accuracy    :", lr_mn.score(X_test, y_test))
```

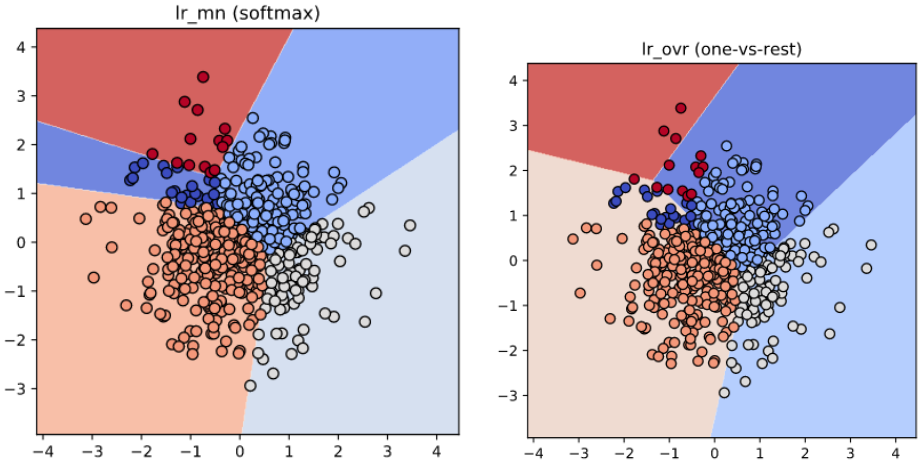
<script.py> output:

OVR training accuracy: 0.9948032665181886
OVR test accuracy : 0.9644444444444444
Softmax training accuracy: 1.0
Softmax test accuracy : 0.9688888888888889

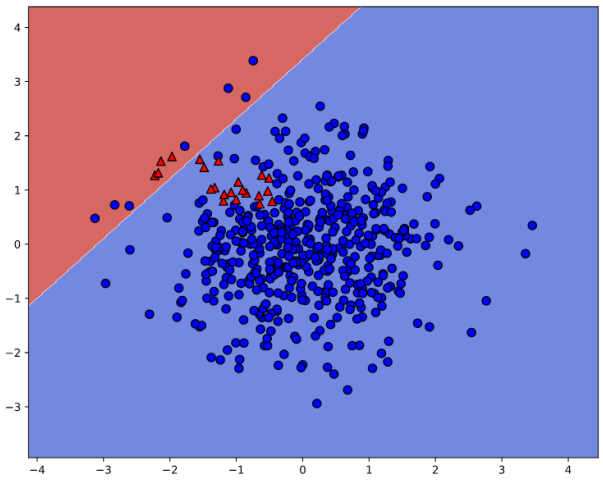
Visualizing multi-class logistic regression

```
# Print training accuracies
print("Softmax     training accuracy:", lr_mn.score
(X_train, y_train))
print("One-vs-rest training accuracy:", lr_ovr.score
(X_train, y_train))

# Create the binary classifier (class 1 vs. rest)
lr_class_1 = LogisticRegression(C=100)
lr_class_1.fit(X_train, y_train==1)

# Plot the binary classifier (class 1 vs. rest)
plot_classifier(X_train, y_train==1, lr_class_1)
```

Using LogReg with strength C=100



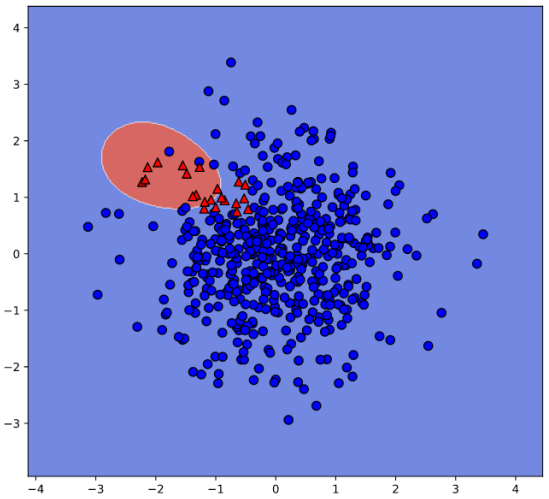
softmax vs one-vs-rest



-> Our LogReg(C=100)

One-vs-rest SVM

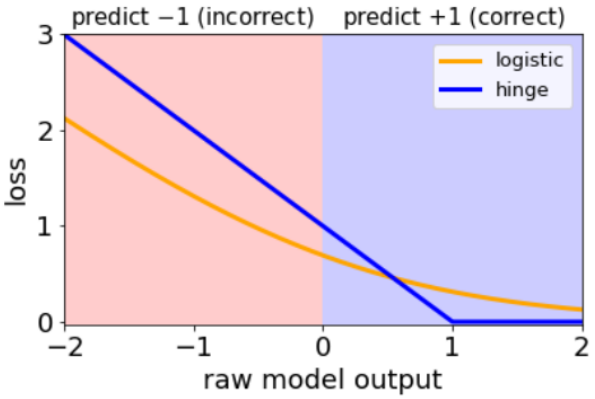
```
# We'll use SVC instead of LinearSVC from now on
from sklearn.svm import SVC

# Create/plot the binary classifier (class 1 vs. rest)
svm_class_1 = SVC()
svm_class_1.fit(X_train, y_train==1)
plot_classifier(X_train, y_train==1, svm_class_1)
```



The non-linear SVM works fine with one-vs-rest on this dataset because it learns to "surround" class 1.

Support Vector Machines (SVM)

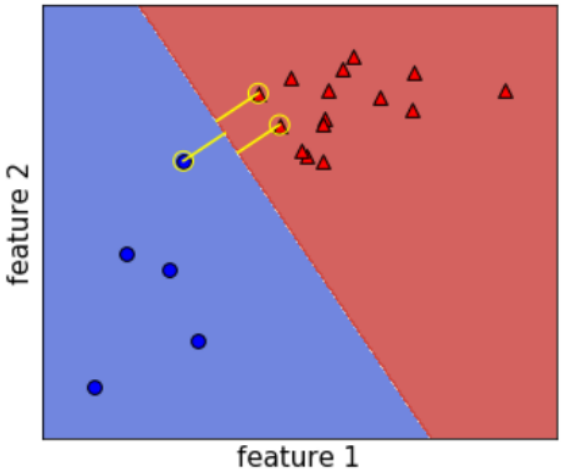
- linear classifier
- trained using the hinge loss and L2 (Ridge) regularization



- a training example not in the flat part of the loss diagram
- an example that is incorrectly classified or close to the boundary
- if not SV, removing it has no effect on the model
- having a small number of SV makes kernel SVMs really fast

Max-margin viewpoint

- SVM maximizes the “margin” for linearly separable datasets
- Margin = distance from the boundary to the closest points



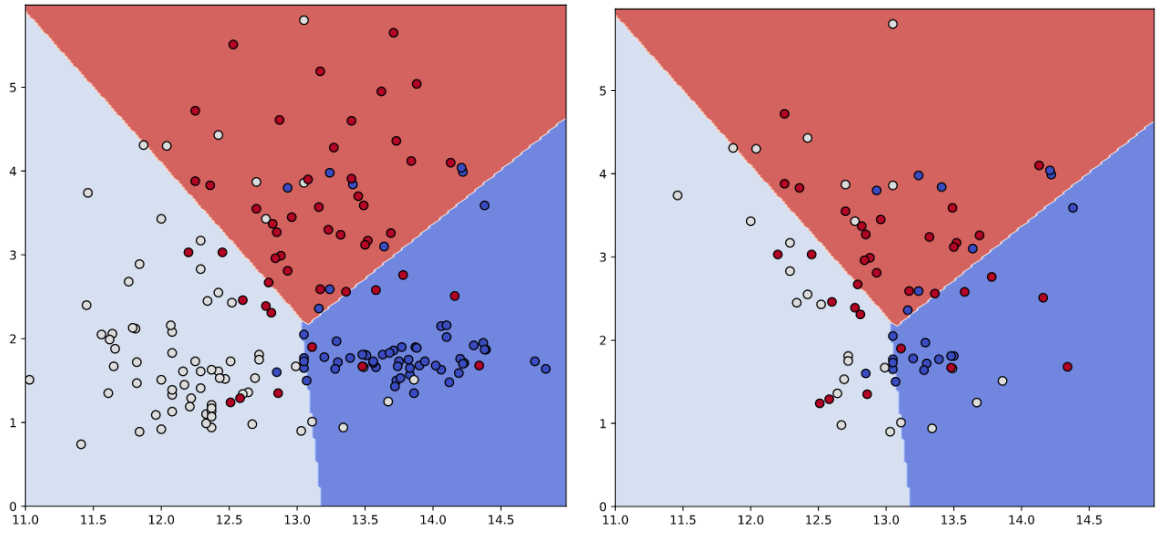
Margin of the SVM

“All incorrectly classified points are support vectors”

Effect of removing examples

```
# Train a linear SVM
svm = SVC(kernel="linear")
svm.fit(X,y)
plot_classifier(X, y, svm, lims=(11,15,0,6))
# Make a new data set keeping only the support vectors
print("Number of original examples", len(X))
print("Number of support vectors", len(svm.support_))
X_small = X[svm.support_]
y_small = y[svm.support_]
# Train a new SVM using only the support vectors
svm_small = SVC(kernel="linear")
svm_small.fit(X,y)
plot_classifier(X_small, y_small, svm_small, lims=(11,15,0,6))
```

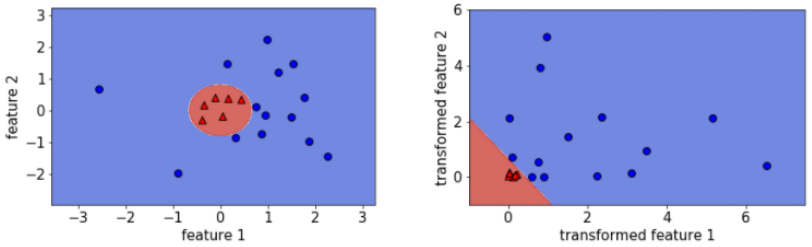
svm.support_ = number of support vectors



Original dataset (left), Dataset after removing examples (right)

Kernel SVMs

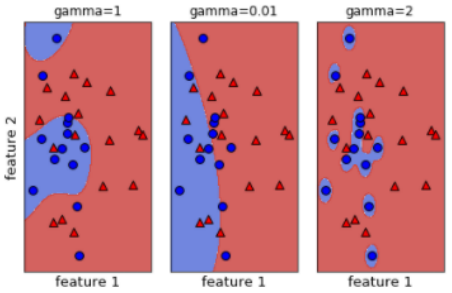
-> **Transforming features** (*example*, squaring original feature)



$$\text{transformed feature} = (\text{original feature})^2$$

Transformed feature can be easily classified

```
from sklearn.svm import SVC

svm = SVC(gamma=2)    # default is kernel="rbf"
```



- larger `gamma` leads to more complex boundaries

```
# Instantiate an RBF SVM
svm = SVC()
# Instantiate the GridSearchCV object and run the search
parameters = {'gamma':[0.00001, 0.0001, 0.001, 0.01, 0.1]}
searcher = GridSearchCV(svm, parameters)
searcher.fit(X,y)
# Report the best parameters
print("Best CV params", searcher.best_params_)
```

Using Grid search to find best CV params

<script.py> output:

Best CV params {'gamma': 0.001}

Tuning gamma and C with Grid Search

```
# Instantiate an RBF SVM
svm = SVC()
# Instantiate the GridSearchCV object and run the search
parameters = {'C':[0.1, 1, 10], 'gamma':[0.00001, 0.0001, 0.001, 0.01, 0.1]}
searcher = GridSearchCV(svm, parameters)
searcher.fit(X_train, y_train)
# Report the best parameters and the corresponding score
print("Best CV params", searcher.best_params_)
print("Best CV accuracy", searcher.best_score_)
# Report the test accuracy using these best parameters
print("Test accuracy of best grid search hypers:", searcher.score(X_test, y_test))
```

<script.py> output:
Best CV params {'C': 10, 'gamma': 0.0001}
Best CV accuracy 0.9988864142538976
Test accuracy of best grid search hypers: 0.9988876529477196

Comparing logistic regression and SVM (and beyond)

Logistic regression:

- Is a linear classifier
- Can use with kernels, but slow
- Outputs meaningful probabilities
- Can be extended to multi-class
- All data points affect fit
- L2 or L1 regularization

Support vector machine (SVM):

- Is a linear classifier
- Can use with kernels, and fast
- Does not naturally output probabilities
- Can be extended to multi-class
- Only "support vectors" affect fit
- Conventionally just L2 regularization

Logistic regression in sklearn:

- `linear_model.LogisticRegression`

Key hyperparameters in sklearn:

- `C` (inverse regularization strength)
- `penalty` (type of regularization)
- `multi_class` (type of multi-class)

SVM in sklearn:

- `svm.LinearSVC` and `svm.SVC`

Key hyperparameters in sklearn:

- `C` (inverse regularization strength)
- `kernel` (type of kernel)
- `gamma` (inverse RBF smoothness)

SGDClassifier

`SGDClassifier` : scales well to large datasets

```
from sklearn.linear_model import SGDClassifier

logreg = SGDClassifier(loss='log')

linsvm = SGDClassifier(loss='hinge')
```

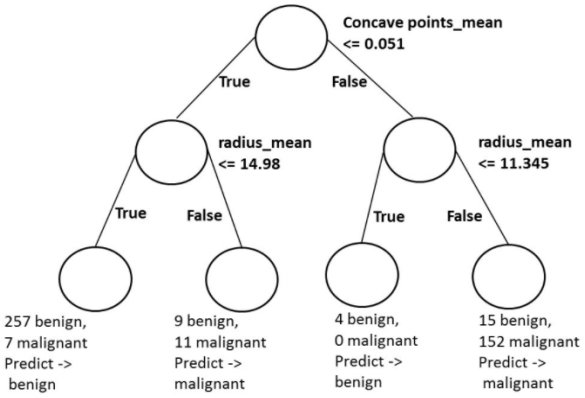
- `SGDClassifier` hyperparameter `alpha` is like `1/C`

```
# We set random_state=0 for reproducibility
linear_classifier = SGDClassifier(random_state=0)
# Instantiate the GridSearchCV object and run the search
parameters = {'alpha':[0.00001, 0.0001, 0.001, 0.01, 0.1, 1],
              'loss':['hinge','log'], 'penalty':['l1','l2']}
searcher = GridSearchCV(linear_classifier, parameters, cv=10)
searcher.fit(X_train, y_train)
# Report the best parameters and the corresponding score
print("Best CV params", searcher.best_params_)
print("Best CV accuracy", searcher.best_score_)
print("Test accuracy of best grid search hypers:", searcher.score(X_test, y_test))
```

One advantage of SGDClassifier is that it's very fast - this would have taken a lot longer with LogisticRegression or LinearSVC.

We compare hinge and log, l1 and l2

Tree-Based Models



Example of decision tree diagram

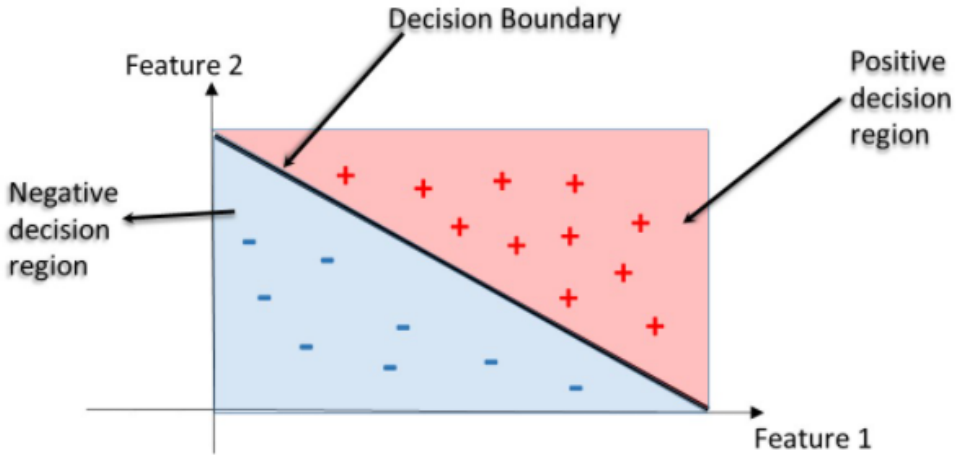
```
# Import DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier
# Import train_test_split
from sklearn.model_selection import train_test_split
# Import accuracy_score
from sklearn.metrics import accuracy_score
# Split dataset into 80% train, 20% test
X_train, X_test, y_train, y_test= train_test_split(X, y,
                                                   test_size=0.2,
                                                   stratify=y,
                                                   random_state=1)

# Instantiate dt
dt = DecisionTreeClassifier(max_depth=2, random_state=1)
```

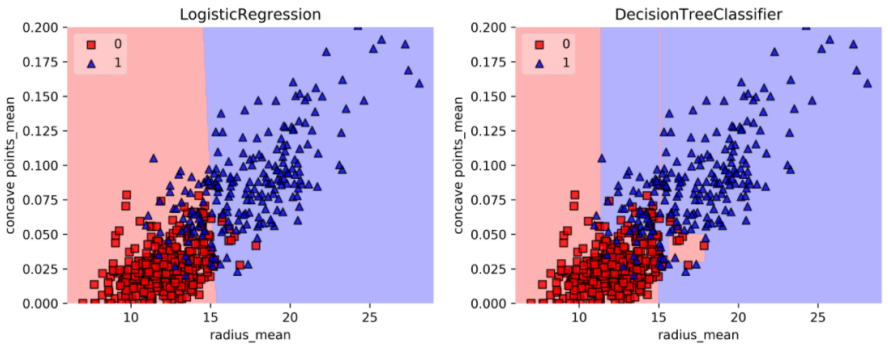
```
# Fit dt to the training set
dt.fit(X_train,y_train)

# Predict test set labels
y_pred = dt.predict(X_test)
# Evaluate test-set accuracy
accuracy_score(y_test, y_pred)
```

```
0.90350877192982459
```

Decision region - region in the feature space where all instances are assigned to one class label

Decision boundary - surface separating different decision regions





Decision regions: CART vs Linear model

```
# Import DecisionTreeClassifier from sklearn.tree
from sklearn.tree import DecisionTreeClassifier
# Instantiate a DecisionTreeClassifier 'dt' with a maximum depth of 6
dt = DecisionTreeClassifier(max_depth=6, random_state=SEED)
# Fit dt to the training set
dt.fit(X_train, y_train)
# Predict test set labels
y_pred = dt.predict(X_test)
print(y_pred[0:5])
```

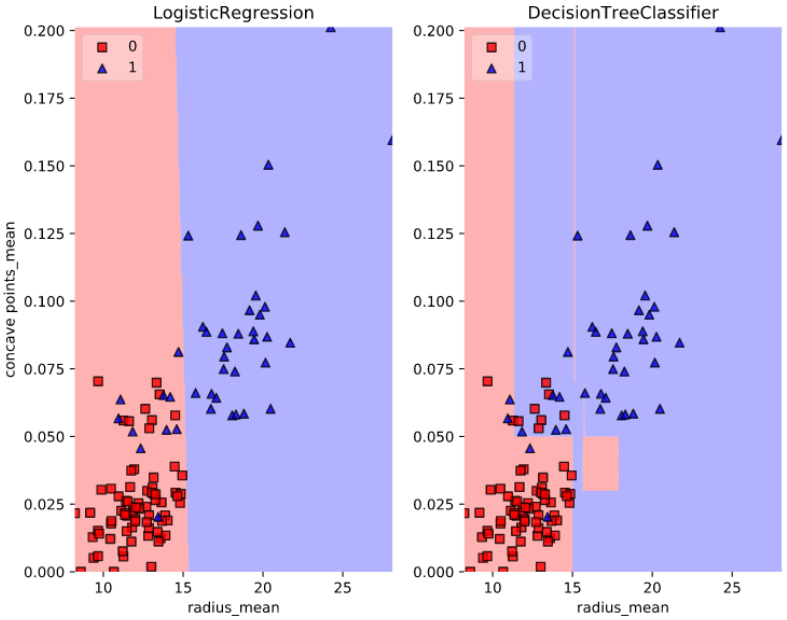
Decision tree with max depth=6, then fit and predict

```
# Import accuracy_score
from sklearn.metrics import accuracy_score
# Predict test set labels
y_pred = dt.predict(X_test)
# Compute test set accuracy
acc = accuracy_score(y_test, y_pred)
print("Test set accuracy: {:.2f}".format(acc))
```

Measure the model accuracy (dt model from last exercise)

```
# Import LogisticRegression from sklearn.linear_model
from sklearn.linear_model import LogisticRegression
# Instatiate logreg
logreg = LogisticRegression(random_state=1)
# Fit logreg to the training set
logreg.fit(X_train, y_train)
# Define a list called clfs containing the two classifiers
logreg and dt
clfs = [logreg, dt]
# Review the decision regions of the two classifiers
plot_labeled_decision_regions(X_test, y_test, clfs)
```

LogReg vs DecisionTree (compare by plotting graph)

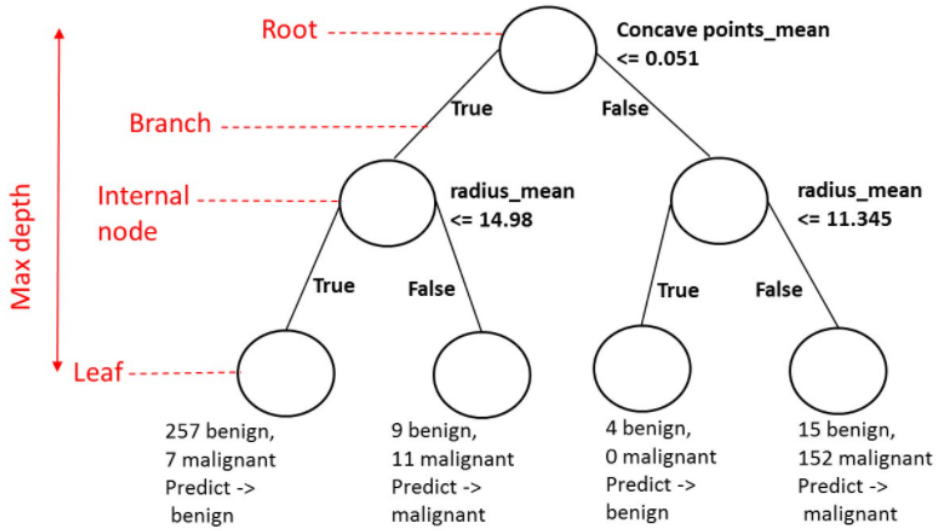


Nodes of decision tree

Root - no parent node, “giving rise to 2 children nodes”

Internal node - one parent node, “giving rise to another 2 children nodes”

Leaf: one parent node, “no children node” → prediction

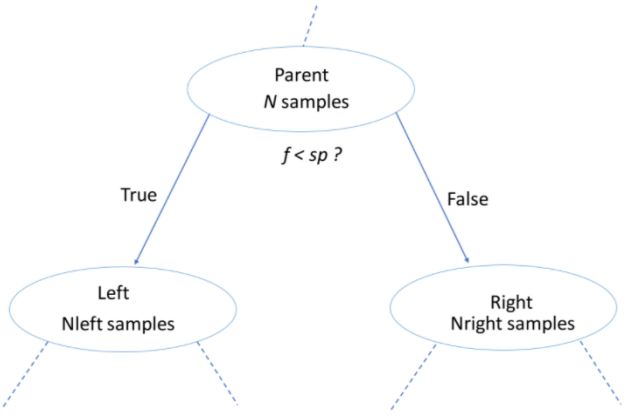


Structure of decision tree

Information Gain (IG)

Information gain calculates the reduction in entropy or surprise from transforming a dataset in some way. A larger information gain suggests a lower entropy group or groups of samples, and hence less surprise.

Entropy quantifies how much information there is in a random variable, or more specifically its probability distribution. A skewed distribution has a low entropy, whereas a distribution where events have equal probability has a larger entropy.



$$IG(\underbrace{f}_{feature}, \underbrace{sp}_{split-point}) = I(parent) - \left(\frac{N_{left}}{N} I(left) + \frac{N_{right}}{N} I(right)\right)$$

Criteria to measure the impurity of a node I(node):

- gini index
- entropy

Classification-tree learning

- nodes are grown recursively
- at each node, split the data based on: feature “f” and split-point “sp” to maximize IG(node)
- If IG(node) =0, declare the node a leaf

```
# Import DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier
# Import train_test_split
from sklearn.model_selection import train_test_split
# Import accuracy_score
from sklearn.metrics import accuracy_score
# Split dataset into 80% train, 20% test
X_train, X_test, y_train, y_test= train_test_split(X, y,
                                                   test_size=0.2,
                                                   stratify=y,
                                                   random_state=1)
# Instantiate dt, set 'criterion' to 'gini'
dt = DecisionTreeClassifier(criterion='gini', random_state=1)
```

```
# Fit dt to the training set
dt.fit(X_train,y_train)

# Predict test-set labels
y_pred= dt.predict(X_test)

# Evaluate test-set accuracy
accuracy_score(y_test, y_pred)
```

```
0.92105263157894735
```

“When an internal node is split, the split is performed in such a way so that information gain is minimized.”

```
# Import DecisionTreeClassifier from sklearn.tree
from sklearn.tree import DecisionTreeClassifier
# Instantiate dt_entropy, set 'entropy' as the information criterion
dt_entropy = DecisionTreeClassifier(max_depth=8, criterion='entropy', random_state=1)
# Fit dt_entropy to the training set
dt_entropy.fit(X_train, y_train)
```

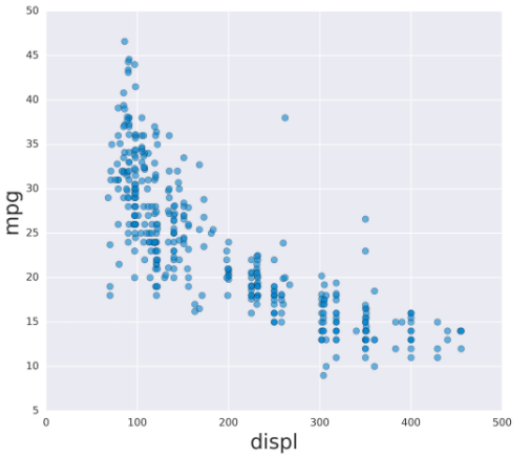
Using ‘entropy’ as a criterion of a decision tree classifier

```
# Import accuracy_score from sklearn.metrics
from sklearn.metrics import accuracy_score
# Use dt_entropy to predict test set labels
y_pred = dt_entropy.predict(X_test)
# Evaluate accuracy_entropy
accuracy_entropy = accuracy_score(y_test, y_pred)
# Print accuracy_entropy
print('Accuracy achieved by using entropy: ', accuracy_entropy)
# Print accuracy_gini
print('Accuracy achieved by using the gini index: ', accuracy_gini)
```

Comparing accuracy of the output between entropy and gini criterions

Auto-mpg dataset

	mpg	displ	hp	weight	accel	origin	size
0	18.0	250.0	88	3139	14.5	US	15.0
1	9.0	304.0	193	4732	18.5	US	20.0
2	36.1	91.0	60	1800	16.4	Asia	10.0
3	18.5	250.0	98	3525	19.0	US	15.0
4	34.3	97.0	78	2188	15.8	Europe	10.0
5	32.9	119.0	100	2615	14.8	Asia	10.0



auto-mpg with one feature

```
# Import DecisionTreeRegressor
from sklearn.tree import DecisionTreeRegressor
# Import train_test_split
from sklearn.model_selection import train_test_split
# Import mean_squared_error as MSE
from sklearn.metrics import mean_squared_error as MSE
# Split data into 80% train and 20% test
X_train, X_test, y_train, y_test= train_test_split(X, y,
                                                   test_size=0.2,
                                                   random_state=3)
# Instantiate a DecisionTreeRegressor 'dt'
dt = DecisionTreeRegressor(max_depth=4,
                           min_samples_leaf=0.1,
                           random_state=3)
```

```
# Fit 'dt' to the training-set
dt.fit(X_train, y_train)
# Predict test-set labels
y_pred = dt.predict(X_test)
# Compute test-set MSE
mse_dt =  MSE(y_test, y_pred)
# Compute test-set RMSE
rmse_dt = mse_dt**(1/2)
# Print rmse_dt
print(rmse_dt)
```

```
5.1023068889
```

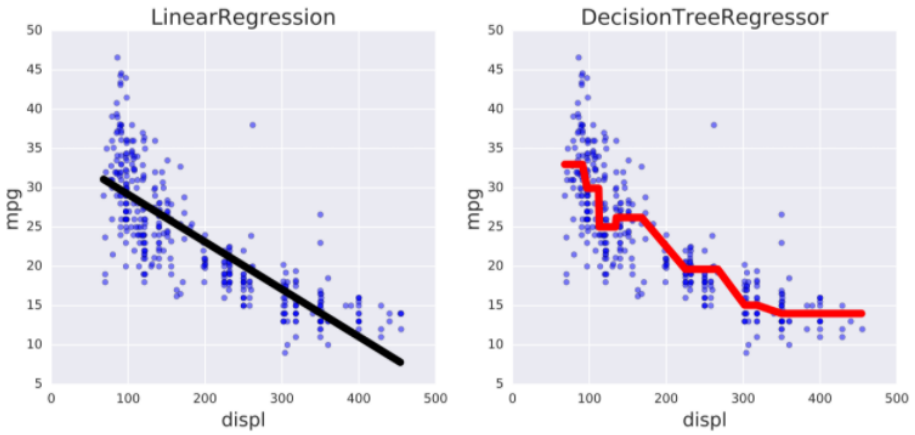
Information Criterion for regression-tree

$$I(\text{node}) = \underbrace{\text{MSE(node)}}_{mean-squared-error} = \frac{1}{N_{node}} \sum_{i \in node} \left(y^{(i)} - \hat{y}_{node}\right)^2$$

$$\underbrace{\hat{y}_{node}}_{mean-target-value} = \frac{1}{N_{node}} \sum_{i \in node} y^{(i)}$$

$$\hat{y}_{pred}(\text{leaf}) = \frac{1}{N_{leaf}} \sum_{i \in leaf} y^{(i)}$$

Prediction



LinearRegression vs Regression-tree

```
# Import DecisionTreeRegressor from sklearn.tree
from sklearn.tree import DecisionTreeRegressor
# Instantiate dt
dt = DecisionTreeRegressor(max_depth=8,
                           min_samples_leaf=0.13,
                           random_state=3)
# Fit dt to the training set
dt.fit(X_train, y_train)
```

Training regression-tree

```
# Import mean_squared_error from sklearn.metrics as MSE
from sklearn.metrics import mean_squared_error as MSE
# Compute y_pred
y_pred = dt.predict(X_test)
# Compute mse_dt
mse_dt = MSE(y_test, y_pred)
# Compute rmse_dt (root mean squared)
rmse_dt = mse_dt**(1/2)
# Print rmse_dt
print("Test set RMSE of dt: {:.2f}".format(rmse_dt))
```

Evaluate the rmse (root mean squared error)

<script.py> output:

Test set RMSE of dt: 4.37

```
# Predict test set labels
y_pred_lr = lr.predict(X_test)
# Compute mse_lr
mse_lr = MSE(y_test, y_pred_lr)
# Compute rmse_lr
rmse_lr = mse_lr**(1/2)
# Print rmse_lr
print('Linear Regression test set RMSE: {:.2f}'.format(rmse_lr))
# Print rmse_dt
print('Regression Tree test set RMSE: {:.2f}'.format(rmse_dt))
```

Linear regression vs Regression-tree

<script.py> output:

Linear Regression test set RMSE: 5.10

Regression Tree test set RMSE: 4.37

The Bias-Variance Tradeoff

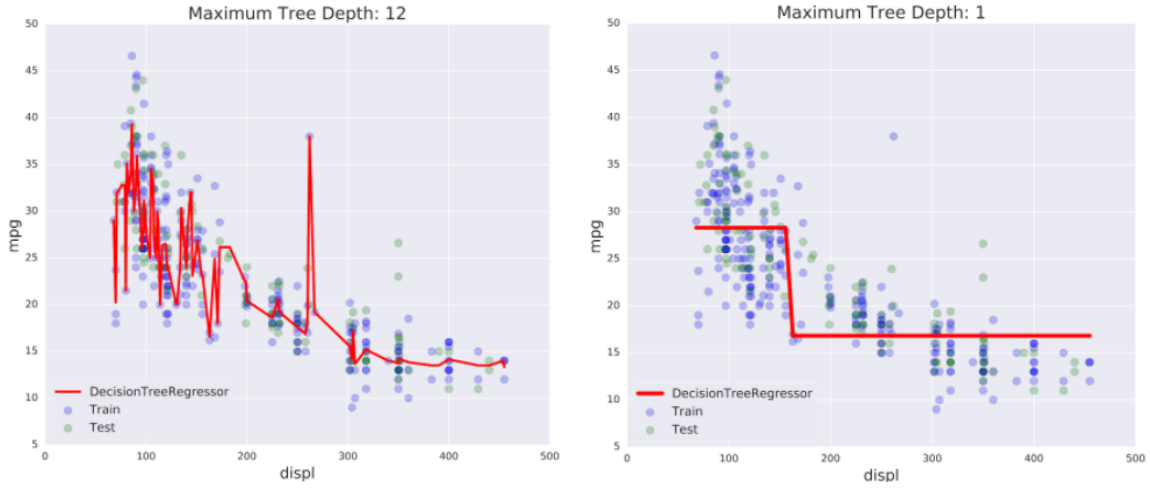
Generalization Error

Goals of Supervised learning:

- find model f_hat that f_hat = f (approx)
- f_hat can be logreg, linearReg, decision_tree, NN, …
- discard noise as much as possible
- END GOAL: f_hat should achieve a low prediction error on unseen datasets

Difficulties in approximating f

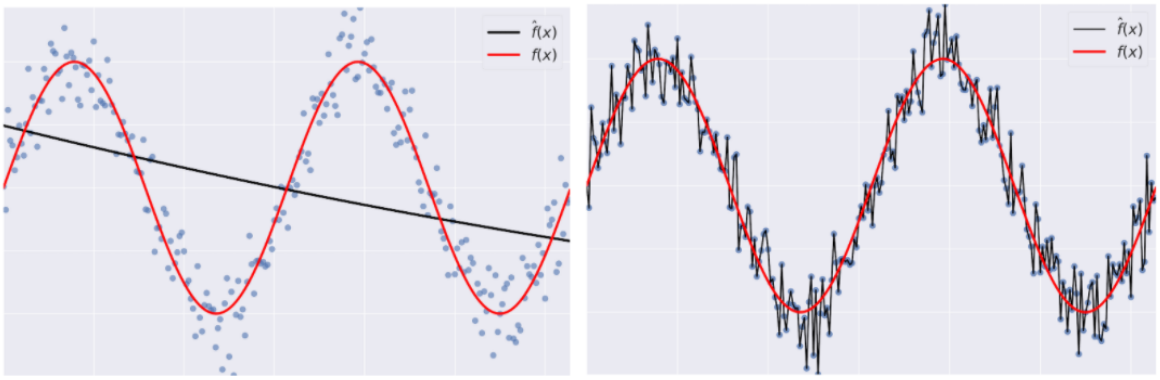
- Overfitting: f_hat fits the training set noise
- Underfitting: f_hat is not flexible enough to approximate f



Overfitting (left), Underfitting (right)

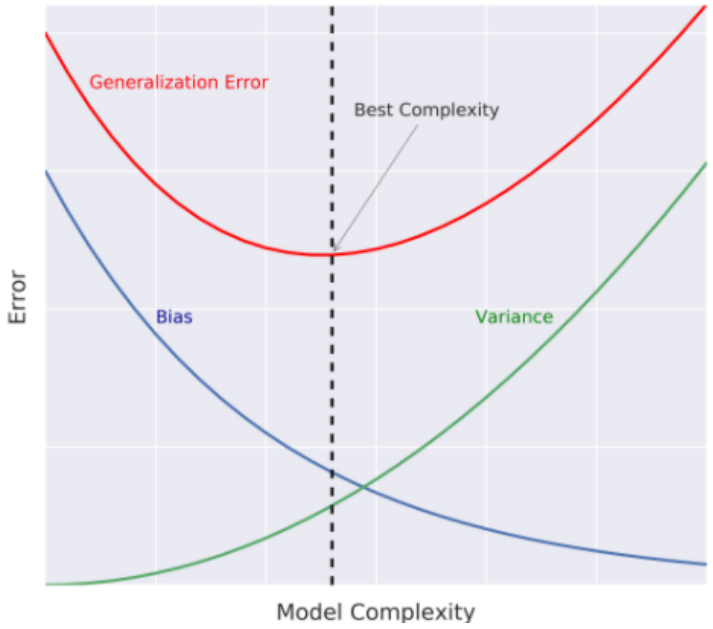
Generalization error: does f_hat generalize well on unseen data?

- f_hat = $bias^2 + variance + irreducible\ error$
- Bias: error term that can tell, on average, how much f_hat != f
- Variance: tells how much f_hat is inconsistent over different training sets

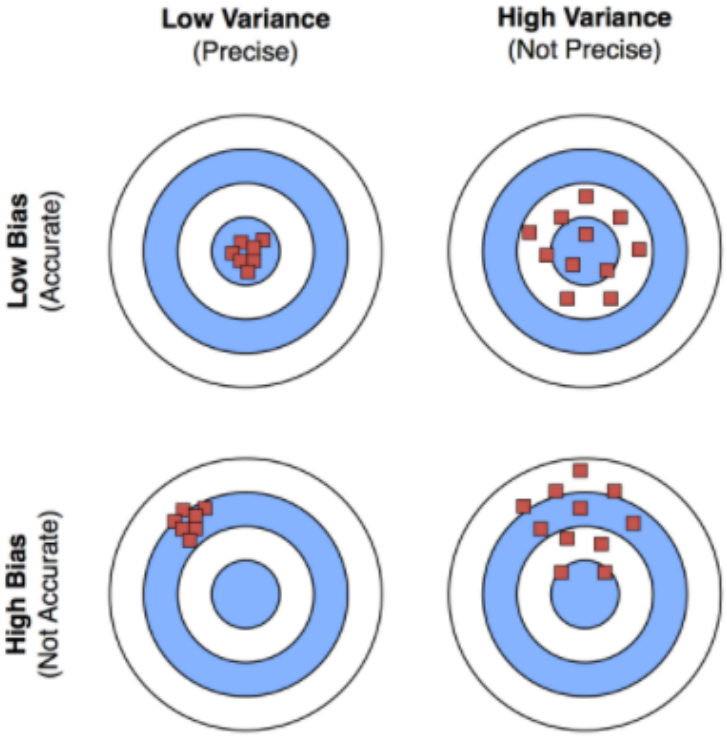


Bias (left), Variance (right)

Model complexity: sets the flexibility of f_hat
→ e.g., maximum tree depth, minimum samples per leaf, etc.



Bias-Variance tradeoff



Bias-Variance tradeoff: Visualization

Diagnose bias and variance problems

Estimating generalization error:

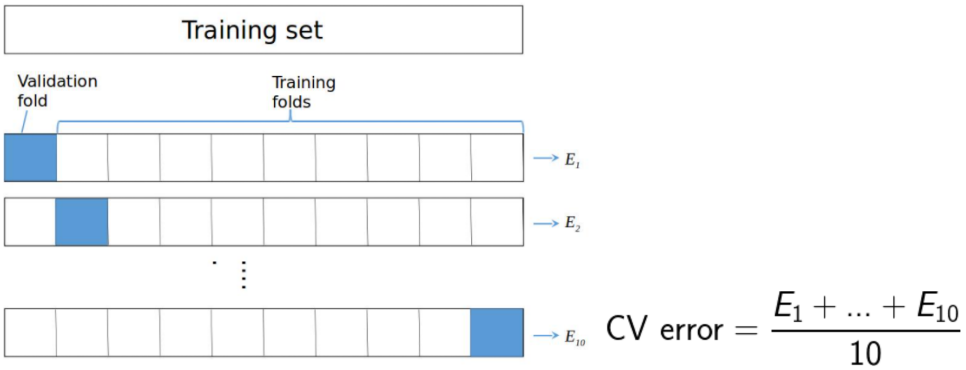
- How do we estimate the generalization error of a model?
- Cannot be done directly because:
  - f is unknown
  - usually we only have 1 dataset
  - noise is unpredictable

Solution:

- split the dataset into train and test
- fit f_hat into training set
- evaluate the error of f_hat on the unseen test set
- ***generalization error of f_hat = test set error of f_hat***

Better model evaluation with Cross-Validation

- Test set should not be touched until we are confident about f_hat's performance
- Evaluating f_hat on training set: biased estimate, f_hat has already seen all training points (what's the point of doing this?!)
- Solution → Cross-Validation (CV)
  - k-Fold CV
  - Hold-out CV



k-Fold CV

→ Diagnose **Variance problem**

- If f_hat suffers from **high variance** = ***CV error of f_hat > training set error of f_hat***
- f_hat is overfit, to remedy that
  - decrease model complexity
  - e.g., decrease max depth, increase min samples per leaf, etc.
  - gather more data

→ Diagnose **Bias problem**

- If f_hat suffers from **high bias** =>
- ***CV error of f_hat = training set error of f_hat >> desired error***
- f_hat is underfit the training set, to remedy that
  - increase model complexity
  - e.g., increase max depth, decrease min samples per leaf, etc.
  - gather more relevant features

k-Fold CV in sklearn on the Auto dataset

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as MSE
from sklearn.model_selection import cross_val_score
# Set seed for reproducibility
SEED = 123
# Split data into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X,y,
                                                    test_size=0.3,
                                                    random_state=SEED)
# Instantiate decision tree regressor and assign it to 'dt'
dt = DecisionTreeRegressor(max_depth=4,
                           min_samples_leaf=0.14,
                           random_state=SEED)
```

```
# Evaluate the list of MSE ontained by 10-fold CV
# Set n_jobs to -1 in order to exploit all CPU cores in computation
MSE_CV = - cross_val_score(dt, X_train, y_train, cv= 10,
                           scoring='neg_mean_squared_error',
                           n_jobs = -1)
# Fit 'dt' to the training set
dt.fit(X_train, y_train)
# Predict the labels of training set
y_predict_train = dt.predict(X_train)
# Predict the labels of test set
y_predict_test = dt.predict(X_test)
```

```
# CV MSE
print('CV MSE: {:.2f}'.format(MSE_CV.mean()))
```

```
CV MSE: 20.51
```

```
# Training set MSE
print('Train MSE: {:.2f}'.format(MSE(y_train, y_predict_train)))
```

```
Train MSE: 15.30
```

```
# Test set MSE
print('Test MSE: {:.2f}'.format(MSE(y_test, y_predict_test)))
```

```
Test MSE: 20.92
```

```
# Import train_test_split from sklearn.model_selection
from sklearn.model_selection import train_test_split
# Set SEED for reproducibility
SEED = 1
# Split the data into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=SEED)
# Instantiate a DecisionTreeRegressor dt
dt = DecisionTreeRegressor(max_depth=4, min_samples_leaf=0.26, random_state=SEED)
```

Instantiate the model

Note that since **cross_val_score** has only the option of evaluating the **negative MSEs**, its output should be multiplied by negative one to obtain the MSEs. The CV RMSE can then be obtained by computing the square root of the average MSE.

```
# Compute the array containing the 10-folds CV MSEs
MSE_CV_scores = - cross_val_score(dt, X_train, y_train, cv=10,
                                  scoring='neg_mean_squared_error',
                                  n_jobs=-1)
# Compute the 10-folds CV RMSE -> average(mean()) -> sqrt
RMSE_CV = (MSE_CV_scores.mean())**(1/2)
# Print RMSE_CV
print('CV RMSE: {:.2f}'.format(RMSE_CV))
```

Compute the RMSE_CV from MSE

```
# Import mean_squared_error from sklearn.metrics as MSE
from sklearn.metrics import mean_squared_error as MSE
# Fit dt to the training set
dt.fit(X_train, y_train)
# Predict the labels of the training set
y_pred_train = dt.predict(X_train)
# Evaluate the training set RMSE of dt
RMSE_train = (MSE(y_train, y_pred_train))**(1/2)
# Print RMSE_train
print('Train RMSE: {:.2f}'.format(RMSE_train))
```

Evaluate the training error

<script.py> output:
Train RMSE: 5.15

Ensemble Learning

CART = Classification And Regression Trees

Advantages of CART

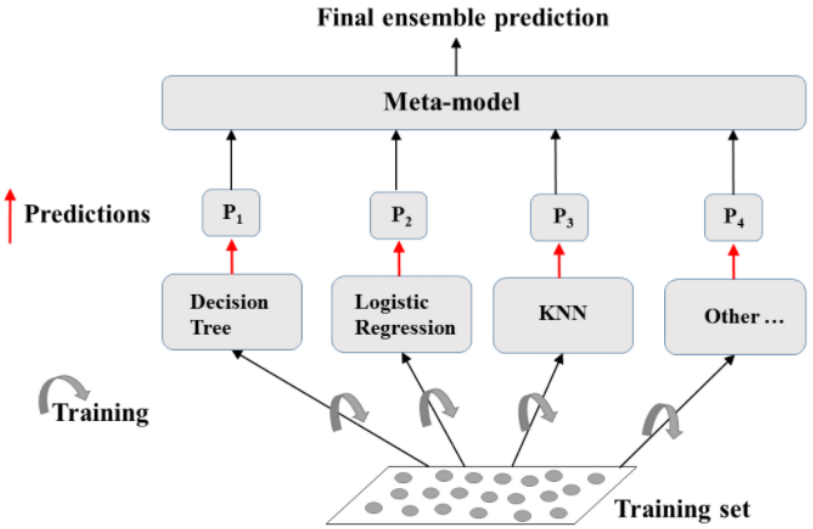
- simple to understand and interpret
- easy to use
- Flexibility: ability to describe non-linear dependencies
- Preprocessing: no need to standardize or normalize features

Limitations of CART

- Classification: can only produce “orthogonal decision boundaries” (มุมฉาก)
- sensitive to small variation in the training sets
- High variance: unconstrained CARTs may overfit the training set
- Solution: “Ensemble Learning”

Ensemble Learning

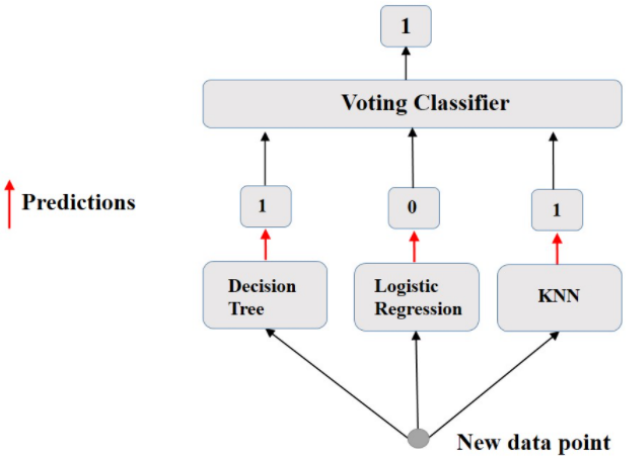
- train different model on the same dataset
- let each model make its predictions
- Meta-model: aggregates predictions of individual models
- Final prediction: more robust and less prone to errors
- Best results: models are skillful in different ways



Visualization of Ensemble learning

Voting Classifier

- binary classification task
- N classifiers make predictions: P1, P2, P3, …, PN with Pi = 0 or 1.
- Meta-model prediction: hard voting



Visualization of Hard voting

Voting classifier in sklearn (Breast-cancer dataset)

```
# Import functions to compute accuracy and split data
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# Import models, including VotingClassifier meta-model
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier as KNN
from sklearn.ensemble import VotingClassifier

# Set seed for reproducibility
SEED = 1
```

```
# Split data into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size= 0.3,
                                                    random_state= SEED)
# Instantiate individual classifiers
lr = LogisticRegression(random_state=SEED)
knn = KNN()
dt = DecisionTreeClassifier(random_state=SEED)
# Define a list called classifier that contains the tuples (classifier_name, classifier)
classifiers = [('Logistic Regression', lr),
               ('K Nearest Neighbours', knn),
               ('Classification Tree', dt)]
```

```
# Iterate over the defined list of tuples containing the classifiers
for clf_name, clf in classifiers:
    #fit clf to the training set
    clf.fit(X_train, y_train)

    # Predict the labels of the test set
    y_pred = clf.predict(X_test)

    # Evaluate the accuracy of clf on the test set
    print('{:s} : {:.3f}'.format(clf_name, accuracy_score(y_test, y_pred)))
```

```
Logistic Regression: 0.947
K Nearest Neighbours: 0.930
Classification Tree: 0.930
```

```
# Instantiate a VotingClassifier 'vc'
vc = VotingClassifier(estimators=classifiers)

# Fit 'vc' to the traing set and predict test set labels
vc.fit(X_train, y_train)
y_pred = vc.predict(X_test)

# Evaluate the test-set accuracy of 'vc'
print('Voting Classifier: {.3f}'.format(accuracy_score(y_test, y_pred)))
```

```
Voting Classifier: 0.953
```

```
# Set seed for reproducibility
SEED=1
# Instantiate lr
lr = LogisticRegression(random_state=SEED)
# Instantiate knn
knn = KNN(n_neighbors=27)
# Instantiate dt
dt = DecisionTreeClassifier(min_samples_leaf=0.13, random_state=SEED)
# Define the list classifiers
classifiers = [('Logistic Regression', lr), ('K Nearest Neighbours', knn), ('Classification Tree', dt)]
```

Define ensemble

```
# Iterate over the pre-defined list of classifiers
for clf_name, clf in classifiers:
    # Fit clf to the training set
    clf.fit(X_train, y_train)
    # Predict y_pred
    y_pred = clf.predict(X_test)
    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    # Evaluate clf's accuracy on the test set
    print('{:s} : {:.3f}'.format(clf_name, accuracy))
```

Evaluating individual classifiers (from last example; 3 classifiers)

<script.py> output:
Logistic Regression : 0.747
K Nearest Neighbours : 0.724
Classification Tree : 0.730

```
# Import VotingClassifier from sklearn.ensemble
from sklearn.ensemble import VotingClassifier
# Instantiate a VotingClassifier vc
vc = VotingClassifier(estimators=classifiers)
# Fit vc to the training set
vc.fit(X_train, y_train)
# Evaluate the test set predictions
y_pred = vc.predict(X_test)
# Calculate accuracy score
accuracy = accuracy_score(y_test, y_pred)
print('Voting Classifier: {:.3f}'.format(accuracy))
```

Voting performance with voting classifiers

<script.py> output:
Voting Classifier: 0.753

Notice how the voting classifier achieves a test set accuracy of 75.3%. This value is greater than that achieved by LogisticRegression

Ensemble methods

Voting Classifier

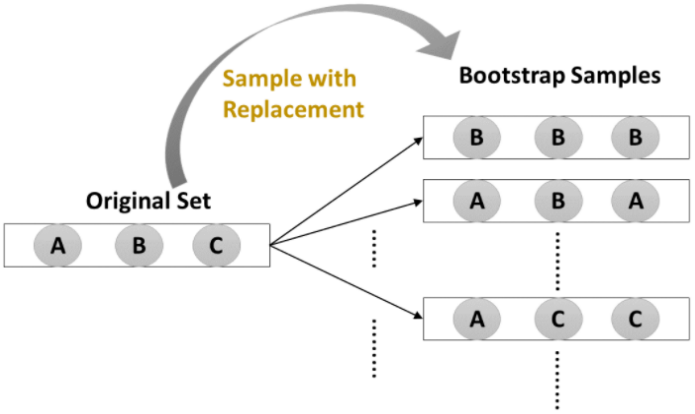
- same training set
- != algorithm

Bagging

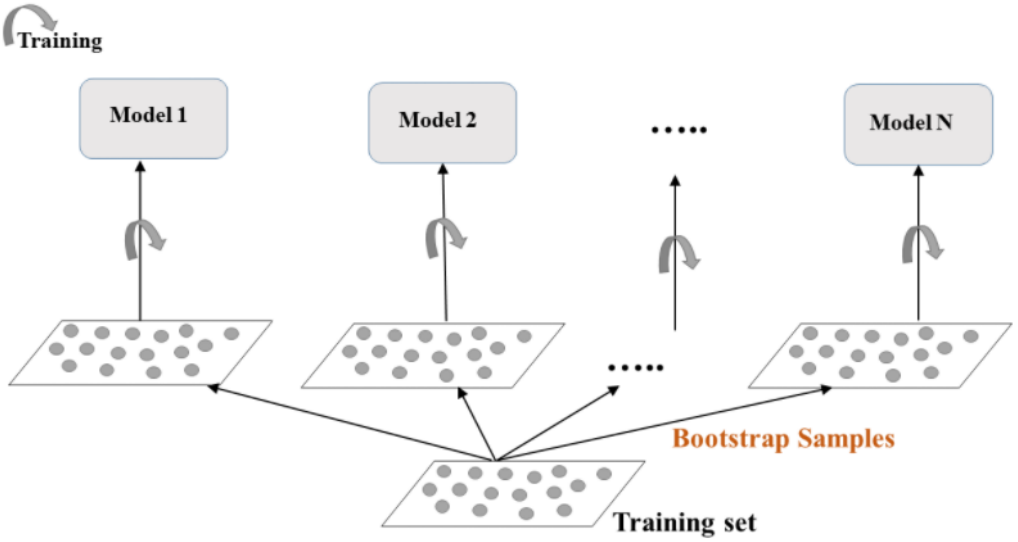
- one algorithm
- != subset of the training set

Bagging = Bootstrap Aggregation

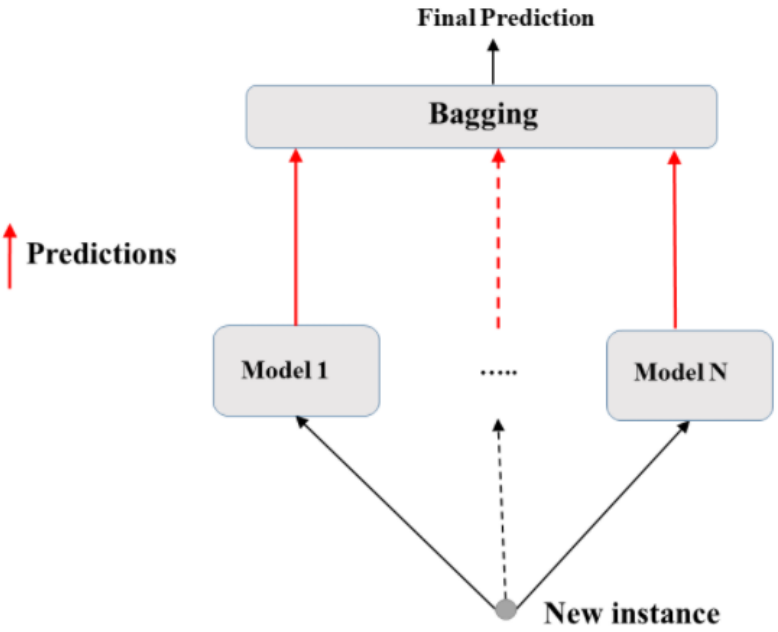
- use a technique = bootstrap
- reduces variance of individual models in the ensemble



Bootstrap



Bagging: training



Bagging: prediction

Bagging classification

- aggregates predictions by majority voting
- BaggingClassifier in sklearn

Bagging Regression

- aggregates predictions through averaging
- BaggingRegression in sklearn

```
# Import models and utility functions
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# Set seed for reproducibility
SEED = 1

# Split data into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    stratify=y,
                                                    random_state=SEED)
```

```
# Instantiate a classification-tree 'dt'
dt = DecisionTreeClassifier(max_depth=4, min_samples_leaf=0.16, random_state=SEED)
# Instantiate a BaggingClassifier 'bc'
bc = BaggingClassifier(base_estimator=dt, n_estimators=300, n_jobs=-1)
# Fit 'bc' to the training set
bc.fit(X_train, y_train)
# Predict test set labels
y_pred = bc.predict(X_test)

# Evaluate and print test-set accuracy
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy of Bagging Classifier: {:.3f}'.format(accuracy))
```

```
Accuracy of Bagging Classifier: 0.936
```

```
# Import DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier
# Import BaggingClassifier
from sklearn.ensemble import BaggingClassifier
# Instantiate dt
dt = DecisionTreeClassifier(random_state=1)
# Instantiate bc
bc = BaggingClassifier(base_estimator=dt, n_estimators=50, random_state=1)
```

Define the BaggingClassifier (esimator is decision tree)

```
# Fit bc to the training set
bc.fit(X_train, y_train)
# Predict test set labels
y_pred = bc.predict(X_test)
# Evaluate acc_test
acc_test = accuracy_score(y_test, y_pred)
print('Test set accuracy of bc: {:.2f}'.format(acc_test))
```

Evaluating bagging performance

<script.py> output:

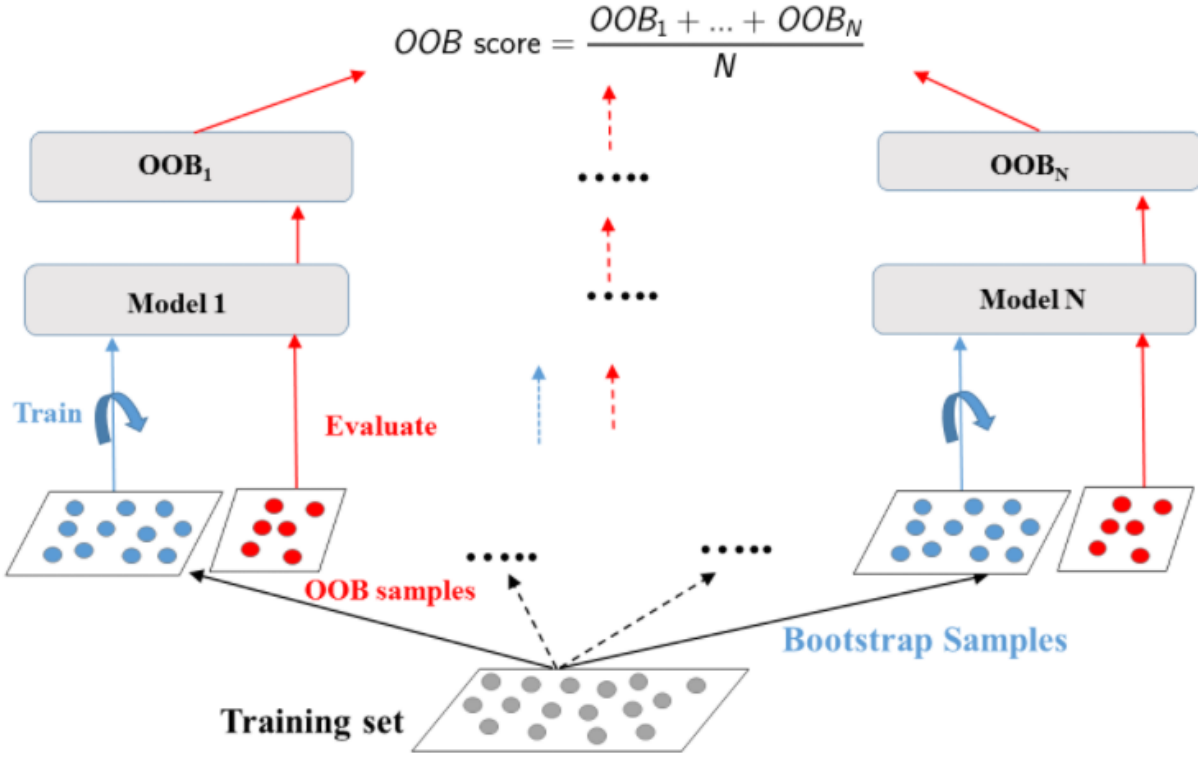
Test set accuracy of bc: 0.71

A single tree dt would have achieved an accuracy of 63% which is 8% lower than bc's accuracy

Out of Bag Evaluation

Bagging → some instance may be sampled several times for one model
→ other instance may not sampled at all
Out of bag (OOB) instance

- On average, for each model, 63% of training instances are sampled
- The remaining 37% constitute the OOB instances



OOB Evaluation

```
# Import models and split utility function
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

# Set seed for reproducibility
SEED = 1

# Split data into 70% train and 30% test
X_train, X_test, y_train, y_test =  train_test_split(X, y, test_size= 0.3,
                                                     stratify= y,
                                                     random_state=SEED)
```

```
# Instantiate a classification-tree 'dt'
dt = DecisionTreeClassifier(max_depth=4,
                            min_samples_leaf=0.16,
                            random_state=SEED)
# Instantiate a BaggingClassifier 'bc'; set oob_score= True
bc = BaggingClassifier(base_estimator=dt, n_estimators=300,
                       oob_score=True, n_jobs=-1)

# Fit 'bc' to the traing set
bc.fit(X_train, y_train)

# Predict the test set labels
y_pred = bc.predict(X_test)
```

```
# Evaluate test set accuracy
test_accuracy = accuracy_score(y_test, y_pred)
# Extract the OOB accuracy from 'bc'
oob_accuracy = bc.oob_score_

# Print test set accuracy
print('Test set accuracy: {:.3f}'.format(test_accuracy))
```

```
Test set accuracy: 0.936
```

```
# Print OOB accuracy
print('OOB accuracy: {:.3f}'.format(oob_accuracy))
```

```
OOB accuracy: 0.925
```

```
# Import DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier
# Import BaggingClassifier
from sklearn.ensemble import BaggingClassifier
# Instantiate dt
dt = DecisionTreeClassifier(min_samples_leaf=8, random_state=1)
# Instantiate bc
bc = BaggingClassifier(base_estimator=dt,
            n_estimators=50,
            oob_score=True,
            random_state=1)
```

Preparing the ground

```
# Fit bc to the training set
bc.fit(X_train, y_train)
# Predict test set labels
y_pred = bc.predict(X_test)
# Evaluate test set accuracy
acc_test = accuracy_score(y_test, y_pred)
# Evaluate OOB accuracy
acc_oob = bc.oob_score_
# Print acc_test and acc_oob
print('Test set accuracy: {:.3f}, OOB accuracy: {:.3f}'.format(acc_test, acc_oob))
```

OOB Score vs Test Set Score

<script.py> output:

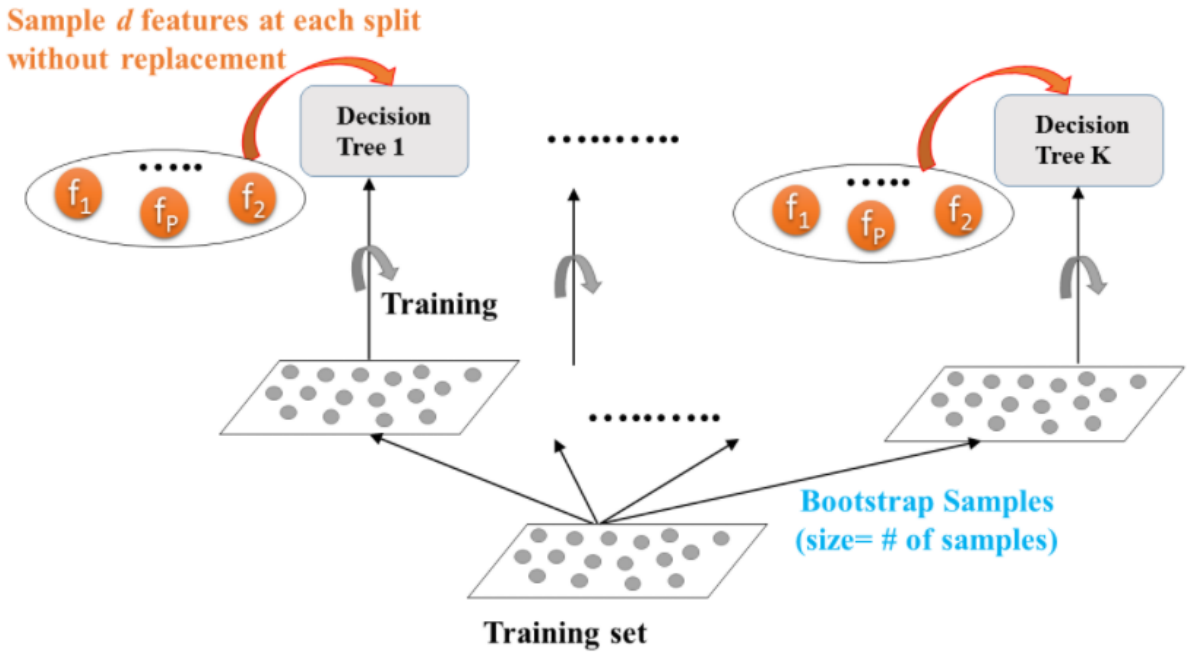
Test set accuracy: 0.698, OOB accuracy: 0.704

The test set accuracy and the OOB accuracy of bc are both roughly equal to 70%!

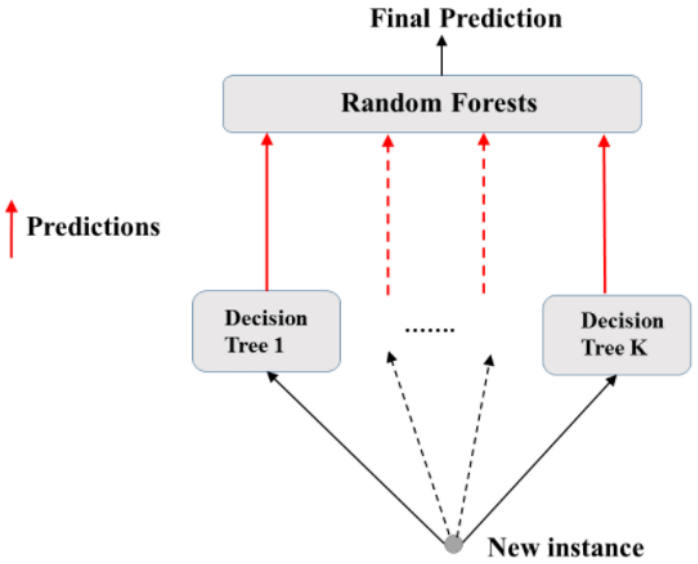
Random Forest (RF)

Bagging → base estimator: decision tree, logReg, NN, etc.
→ each estimator is trained on a distinct bootstrap sample of the training set
→ Estimators use all features for training and prediction

Further diversity with random forest (RF)
→ base estimator: decision tree
→ each estimator is trained on a different bootstrap sample having the same size as the training set
→ RF introduces further randomization in the training of individual trees
→ d features are sampled at each node without replacement
(d < total number of features)



Random forest: training



Random forest: prediction

RF classification

- aggregates predictions by majority voting
- RandomForestClassifier in sklearn

RF regression

- aggregates predictions through averaging
- RandomForestRegressor in sklearn

Random forest regression in sklearn (auto dataset)

```
# Basic imports
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as MSE
# Set seed for reproducibility
SEED = 1

# Split dataset into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.3,
                                                    random_state=SEED)
```

```
# Instantiate a random forests regressor 'rf' 400 estimators
rf = RandomForestRegressor(n_estimators=400,
                                    min_samples_leaf=0.12,
                                    random_state=SEED)
# Fit 'rf' to the training set
rf.fit(X_train, y_train)
# Predict the test set labels 'y_pred'
y_pred = rf.predict(X_test)
```

```
# Evaluate the test set RMSE
rmse_test = MSE(y_test, y_pred)**(1/2)

# Print the test set RMSE
print('Test set RMSE of rf: {:.2f}'.format(rmse_test))
```

```
Test set RMSE of rf: 3.98
```

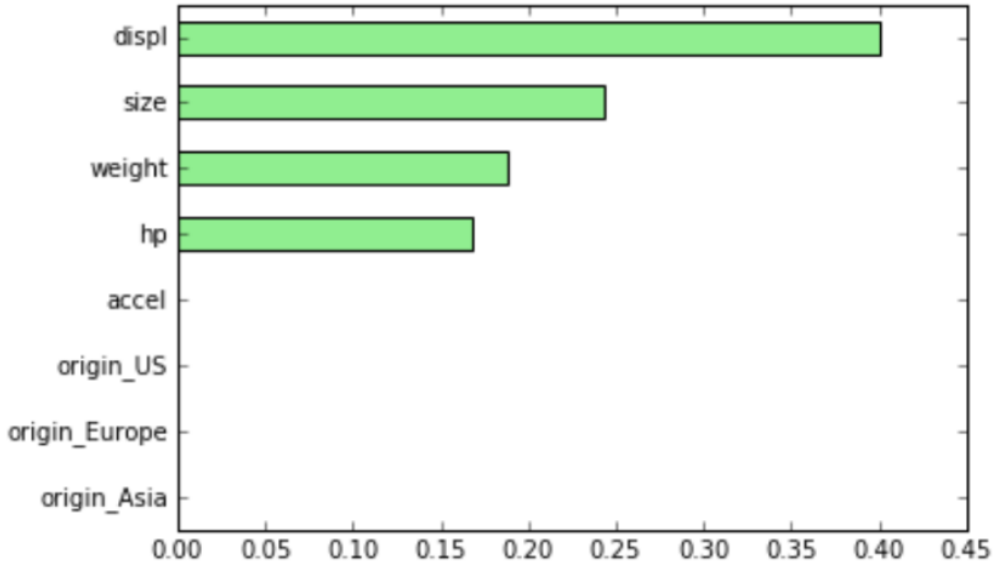
Feature Importance

→ **Tree-based model**: enable measuring the importance of each feature in prediction

In sklearn:

- how much the tree nodes use a particular feature (weighted average) to reduce impurity
- accessed using the attribute “feature_importance_”

```
import pandas as pd
import matplotlib.pyplot as plt

# Create a pd.Series of features importances
importances_rf = pd.Series(rf.feature_importances_, index = X.columns)

# Sort importances_rf
sorted_importances_rf = importances_rf.sort_values()

# Make a horizontal bar plot
sorted_importances_rf.plot(kind='barh', color='lightgreen'); plt.show()
```




```
# Import RandomForestRegressor
from sklearn.ensemble import RandomForestRegressor
# Instantiate rf
rf = RandomForestRegressor(n_estimators=25,
            random_state=2)
# Fit rf to the training set
rf.fit(X_train, y_train)
```

Fitting an RF regressor

```
# Import mean_squared_error as MSE
from sklearn.metrics import mean_squared_error as MSE
# Predict the test set labels
y_pred = rf.predict(X_test)
# Evaluate the test set RMSE
rmse_test = MSE(y_test,y_pred)**(1/2)
# Print rmse_test
print('Test set RMSE of rf: {:.2f}'.format(rmse_test))
```

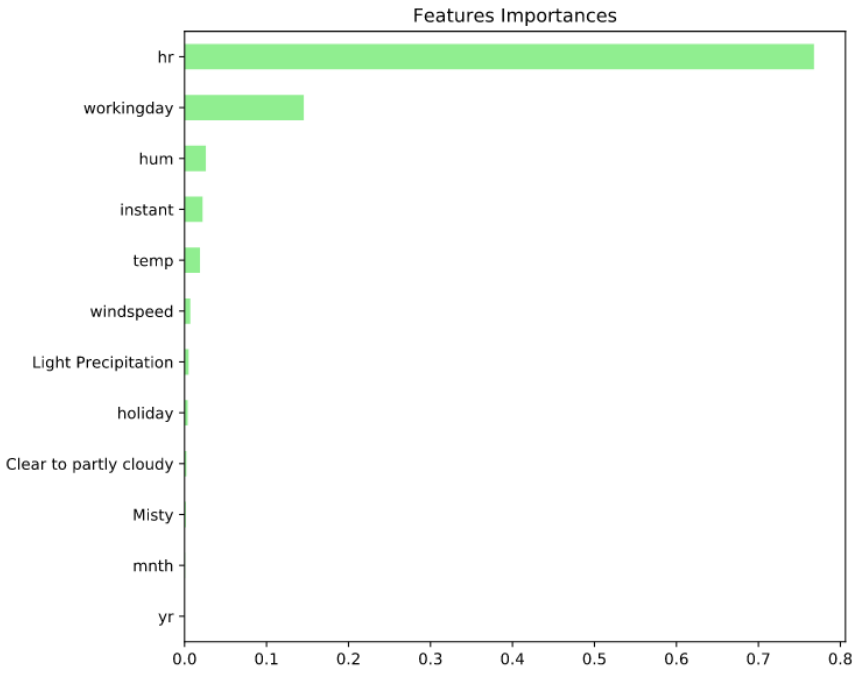
Evaluate the RF regressor

<script.py> output:

Test set RMSE of rf: 51.97

```
# Create a pd.Series of features importances
importances = pd.Series
(data=rf.feature_importances_,
                                index= X_train.columns)
# Sort importances
importances_sorted = importances.sort_values()
# Draw a horizontal barplot of importances_sorted
importances_sorted.plot(kind='barh',
color='lightgreen')
plt.title('Features Importances')
plt.show()
```

Visualizing feature importance



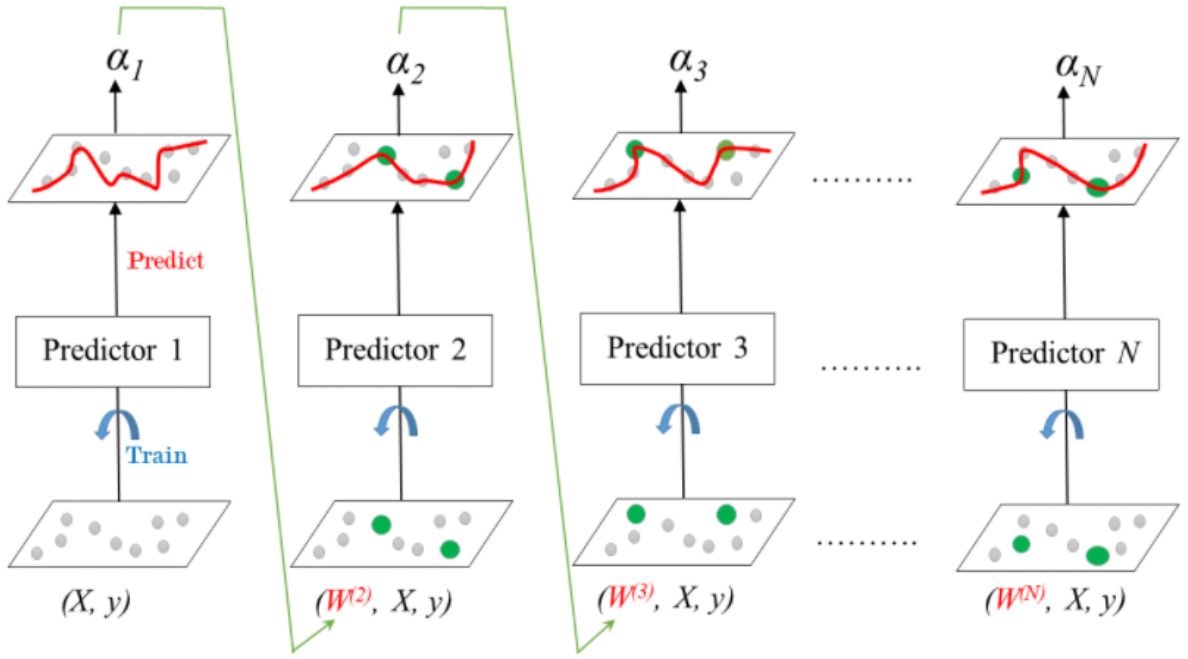
hr and workingday are the most important features according to rf. The importances of these two features add up to more than 90%

Boosting

- Boosting: Ensemble method combining several weak learners to form a strong leaner
- Weak learner: Model doing slightly better than random guessing
- Example of weak learner: Decision Stump (CART whose maximum depth = 1)
- Train an ensemble of predictors sequentially
- Each predictor tries to correct its predecessor
- Most popular boosting methods
    - AdaBoost
    - Gradient boosting

AdaBoost

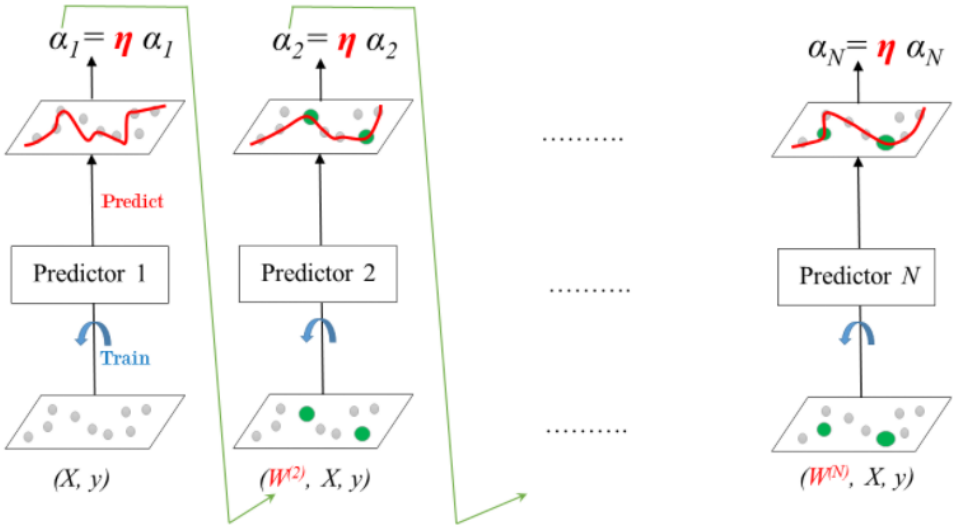
- stands for Adaptive Boosting
- Each predictor pays more attention to the instances wrongly predicted by its predecessor
- Achieved by changing the weight of training instances
- Each predictor is assigned a coefficient α
- α depends on the predictor's training error



AdaBoost: training

Learning Rate

Learning rate: $0 < \eta \leq 1$



AdaBoost classification

- weighted majority voting
- In sklearn: AdaBoostClassifier

AdaBoost regression

- weighted average
- In sklearn: AdaBoostRegressor

```
# Import models and utility functions
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split

# Set seed for reproducibility
SEED = 1

# Split data into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    stratify=y,
                                                    random_state=SEED)
```

```
# Instantiate a classification-tree 'dt'
dt = DecisionTreeClassifier(max_depth=1, random_state=SEED)

# Instantiate an AdaBoost classifier 'adab_clf'
adb_clf = AdaBoostClassifier(base_estimator=dt, n_estimators=100)


# Fit 'adb_clf' to the training set
adb_clf.fit(X_train, y_train)

# Predict the test set probabilities of positive class
y_pred_proba = adb_clf.predict_proba(X_test)[:,1]


# Evaluate test-set roc_auc_score
adb_clf_roc_auc_score = roc_auc_score(y_test, y_pred_proba)
```

```
# Print adb_clf_roc_auc_score
print('ROC AUC score: {:.2f}'.format(adb_clf_roc_auc_score))
```

```
ROC AUC score: 0.99
```

```
# Import DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier
# Import AdaBoostClassifier
from sklearn.ensemble import AdaBoostClassifier
# Instantiate dt
dt = DecisionTreeClassifier(max_depth=2, random_state=1)
# Instantiate ada
ada = AdaBoostClassifier(base_estimator=dt, n_estimators=180, random_state=1)
```

Defining the AdaBoost classifier

```
# Fit ada to the training set
ada.fit(X_train, y_train)
# Compute the probabilities of obtaining the positive class
y_pred_proba = ada.predict_proba(X_test)[:,1]
```

Train the AdaBoost classifier

```
# Import roc_auc_score
from sklearn.metrics import roc_auc_score
# Evaluate test-set roc_auc_score
ada_roc_auc = roc_auc_score(y_test, y_pred_proba)
# Print roc_auc_score
print('ROC AUC score: {:.2f}'.format(ada_roc_auc))
```

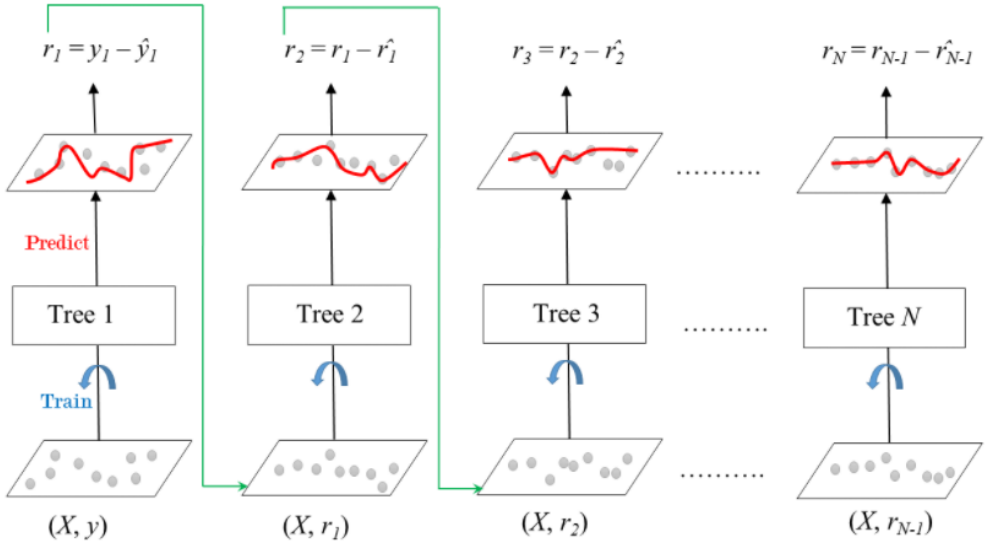
Evaluating the AdaBoost classifier

<script.py> output:
ROC AUC score: 0.71

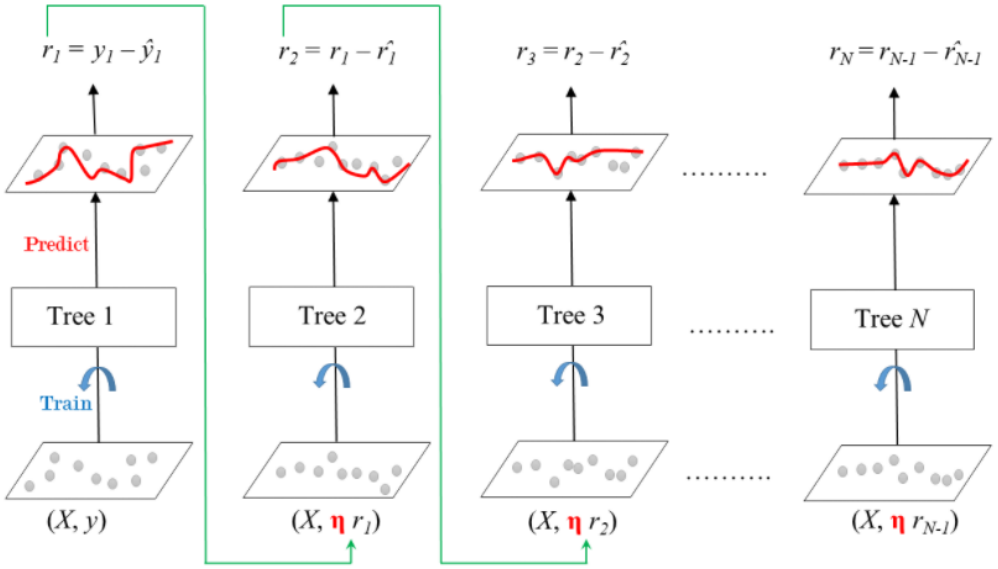
Gradient Boosting (GB)

Gradient boosted tree

- sequential correction of predecessors's errors
- does not tweak the weights of training instances
- fit each predictor is trained using its predecessor's residual errors as labels
- gradient boosted tree: a CART is used as a base learner



Gradient boosted tree for regression: training



Shrinkage

Gradient boosted tree: prediction

Regression:

- $y_{pred} = y_1 + \eta r_1 + \ldots + \eta r_N$
- In sklearn: `GradientBoostingRegressor`.

Classification:

- In sklearn: `GradientBoostingClassifier`.

```
# Import models and utility functions
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error as MSE

# Set seed for reproducibility
SEED = 1

# Split dataset into 70% train and 30% test
X_train, X_test, y_train, y_test = train_test_split(X,y,
                                                    test_size=0.3,
                                                    random_state=SEED)
```

```
# Instantiate a GradientBoostingRegressor 'gbt'
gbt = GradientBoostingRegressor(n_estimators=300, max_depth=1, random_state=SEED)

# Fit 'gbt' to the training set
gbt.fit(X_train, y_train)

# Predict the test set labels
y_pred = gbt.predict(X_test)

# Evaluate the test set RMSE
rmse_test = MSE(y_test, y_pred)**(1/2)

# Print the test set RMSE
print('Test set RMSE: {:.2f}'.format(rmse_test))
```

```
Test set RMSE: 4.01
```

```
# Import GradientBoostingRegressor
from sklearn.ensemble import GradientBoostingRegressor
# Instantiate gb
gb = GradientBoostingRegressor(max_depth=4,
            n_estimators=200,
            random_state=2)
```

Define the GB regressor

```
# Fit gb to the training set
gb.fit(X_train, y_train)
# Predict test set labels
y_pred = gb.predict(X_test)
```

Training and predicting the GB regressor.

```
# Import mean_squared_error as MSE
from sklearn.metrics import mean_squared_error as MSE
# Compute MSE
mse_test = MSE(y_test, y_pred)
# Compute RMSE
rmse_test = mse_test**(1/2)
# Print RMSE
print('Test set RMSE of gb: {:.3f}'.format(rmse_test))
```

Evaluating the RMSE

<script.py> output:
Test set RMSE of gb: 52.065

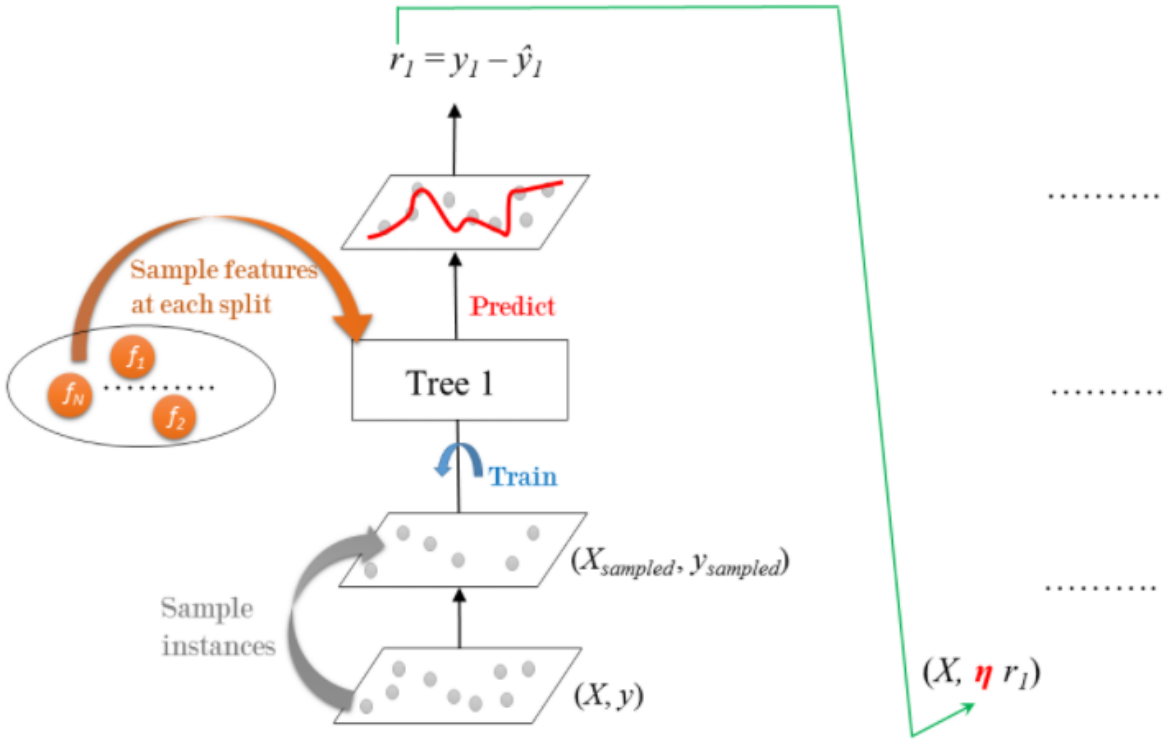
Stochastic Gradient Boosting (SGB)

Cons of gradient boosting

- GB involves an exhaustive search procedure
- Each CART is trained to find the best split points and features
- May lead to CARTs using the same split points and maybe the same features

SGB

- Each tree is trained on a random subset of rows of the training data
- The sampled instances (40%-80% of the training set) are sampled without replacement
- Features are sampled (without replacement) when choosing split points
- Result: further ensemble diversity
- Effect: adding further variance to the ensemble of trees



Stochastic gradient boosting: training

```
# Instantiate a stochastic GradientBoostingRegressor 'sgbt'
sgbt = GradientBoostingRegressor(max_depth=1,
                                 subsample=0.8,
                                 max_features=0.2,
                                 n_estimators=300,
                                 random_state=SEED)

# Fit 'sgbt' to the training set
sgbt.fit(X_train, y_train)

# Predict the test set labels
y_pred = sgbt.predict(X_test)
```

```
# Evaluate test set RMSE 'rmse_test'
rmse_test = MSE(y_test, y_pred)**(1/2)

# Print 'rmse_test'
print('Test set RMSE: {:.2f}'.format(rmse_test))
```

```
Test set RMSE: 3.95
```

```
# Import GradientBoostingRegressor
from sklearn.ensemble import GradientBoostingRegressor
# Instantiate sgbr
sgbr = GradientBoostingRegressor(max_depth=4,
            subsample=0.9,
            max_features=0.75,
            n_estimators=200,
            random_state=2)
```

Regression with SGB

```
# Fit sgbr to the training set
sgbr.fit(X_train, y_train)
# Predict test set labels
y_pred = sgbr.predict(X_test)
```

Train the SGB regressor

```
# Import mean_squared_error as MSE
from sklearn.metrics import mean_squared_error as MSE
# Compute test set MSE
mse_test = MSE(y_test, y_pred)
# Compute test set RMSE
rmse_test = mse_test**(1/2)
# Print rmse_test
print('Test set RMSE of sgbr: {:.3f}'.format(rmse_test))
```

Evaluate the SGB regressor by finding the RMSE

<script.py> output:
Test set RMSE of sgbr: 49.979

The stochastic gradient boosting regressor achieves a lower test set RMSE than the gradient boosting regressor (which was 52.065)

Tuning a CART's Hyperparameters

In machine learning
Parameter → learned from data
CART example: split-point of a node, split-feature of a node
Hyperparameter → not learned from data, set prior to training
CART example: max_depth, min_samples_leaf, splitting criterion, etc.

What is hyperparameter tuning?

- **Problem**: search for a set of optimal hyperparameters for a learning algorithm
- **Solution**: Find a set of optimal hyperparameters that results in an optimal ***model***
- **Optimal model**: yields an optimal ***score***
- **Score**: In sklearn defaults to ***accuracy*** (classification) and R^2 (regression)
- **Cross-validation** is used to estimate the generalization performance

Why tune hyperparameters?

- In sklearn, a model's default hyperparameters are not optimal for all problems
- Hyperparameters should be tuned to obtain the best model performance

Approaches

- Grid search
- Random search
- Bayesian optimization
- Genetic algorithms
- etc.

Grid search cross validation

- Manually set a grid of discrete hyperparameter values.
- Set a metric for scoring model performance.
- Search exhaustively through the grid.
- For each set of hyperparameters, evaluate each model's CV score.
- The optimal hyperparameters are those of the model achieving the best CV score.

Grid search cross validation: example

- Hyperparameters grids:
  - `max_depth` = {2,3,4},
  - `min_samples_leaf` = {0.05, 0.1}
- hyperparameter space = { (2,0.05) , (2,0.1) , (3,0.05), ... }
- CV scores = { $score_{(2,0.05)}$, ... }
- optimal hyperparameters = set of hyperparameters corresponding to the best CV score.

```
# Import DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier

# Set seed to 1 for reproducibility
SEED = 1

# Instantiate a DecisionTreeClassifier 'dt'
dt = DecisionTreeClassifier(random_state=SEED)
```

```
# Print out 'dt's hyperparameters
print(dt.get_params())
```

```
{'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': None,
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'presort': False,
 'random_state': 1,
 'splitter': 'best'}
```

```
# Import GridSearchCV
from sklearn.model_selection import GridSearchCV
# Define the grid of hyperparameters 'params_dt'
params_dt = {
             'max_depth': [3, 4,5, 6],
             'min_samples_leaf': [0.04, 0.06, 0.08],
             'max_features': [0.2, 0.4,0.6, 0.8]
            }
# Instantiate a 10-fold CV grid search object 'grid_dt'
grid_dt = GridSearchCV(estimator=dt,
                       param_grid=params_dt,
                       scoring='accuracy',
                       cv=10,
                       n_jobs=-1)
# Fit 'grid_dt' to the training data
grid_dt.fit(X_train, y_train)
```

```
# Extract best hyperparameters from 'grid_dt'
best_hyperparams = grid_dt.best_params_
print('Best hyerparameters:\n', best_hyperparams)
```

```
Best hyerparameters:
  {'max_depth': 3, 'max_features': 0.4, 'min_samples_leaf': 0.06}
```

```
# Extract best CV score from 'grid_dt'
best_CV_score = grid_dt.best_score_
print('Best CV accuracy'.format(best_CV_score))
```

```
Best CV accuracy: 0.938
```

```
# Extract best model from 'grid_dt'
best_model = grid_dt.best_estimator_

# Evaluate test set accuracy
test_acc = best_model.score(X_test,y_test)

# Print test set accuracy
print("Test set accuracy of best model: {:.3f}".format(test_acc))
```

```
Test set accurac
```

```
# Define params_dt
params_dt = {
    'max_depth': [2,3,4],
    'min_samples_leaf': [0.12,0.14,0.16,0.18]
}
```

Set the tree's hyperparameter grid

```
# Import GridSearchCV
from sklearn.model_selection import GridSearchCV
# Instantiate grid_dt
grid_dt = GridSearchCV(estimator=dt,
                       param_grid=params_dt, # set
                       scoring='roc_auc',
                       cv=5,
                       n_jobs=-1)
```

Search for the optimal tree (using params_dt we set last example)

```
# Import roc_auc_score from sklearn.metrics
from sklearn.metrics import roc_auc_score
# Extract the best estimator (from grid search)
best_model = grid_dt.best_estimator_ # grid_dt had been fit
# Predict the test set probabilities of the positive class
y_pred_proba = best_model.predict_proba(X_test)[:,1]
# Compute test_roc_auc
test_roc_auc = roc_auc_score(y_test, y_pred_proba)
# Print test_roc_auc
print('Test set ROC AUC score: {:.3f}'.format(test_roc_auc))
```

Evaluating the optimal tree

<script.py> output:
Test set ROC AUC score: 0.610

An untuned classification-tree would achieve a ROC AUC score of 0.54!

Tuning RF (Random forest)
RF hyperparameters

- CART hyperparameters
- number of estimators
- boostrap
- etc.

Tuning is expensive
Hyperparameter tuning:

- computationally expensive

- sometimes leads to very slight improvement

Weight the impact of tuning on the whole project

```
# Import RandomForestRegressor
from sklearn.ensemble import RandomForestRegressor

# Set seed for reproducibility
SEED = 1

# Instantiate a random forests regressor 'rf'
rf = RandomForestRegressor(random_state= SEED)
```

```
# Inspect rf' s hyperparameters
rf.get_params()
```

```
{'bootstrap': True,
 'criterion': 'mse',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 10,
 'n_jobs': -1,
 'oob_score': False,
 'random_state': 1,
 'verbose': 0,
 'warm_start': False}
```

```
# Basic imports
from sklearn.metrics import mean_squared_error as MSE
from sklearn.model_selection import GridSearchCV
# Define a grid of hyperparameter 'params_rf'
params_rf = {
              'n_estimators': [300, 400, 500],
              'max_depth': [4, 6, 8],
              'min_samples_leaf': [0.1, 0.2],
              'max_features': ['log2', 'sqrt']
             }
# Instantiate 'grid_rf'
grid_rf = GridSearchCV(estimator=rf,
                       param_grid=params_rf,
                       cv=3,
                       scoring='neg_mean_squared_error',
                       verbose=1,
                       n_jobs=-1)
```

Searching for the best hyperparameters

```
# Fit 'grid_rf' to the training set
grid_rf.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 36 candidates, totalling 108 fits
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:   10.0s
[Parallel(n_jobs=-1)]: Done 108 out of 108 | elapsed:   24.3s finished
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=4,
           max_features='log2', max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_impurity_split=None,
           min_samples_leaf=0.1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators=400, n_jobs=1,
           oob_score=False, random_state=1, verbose=0, warm_start=False)
```

```
# Extract best hyperparameters from 'grid_rf'
best_hyperparams = grid_rf.best_params_

print('Best hyerparameters:\n', best_hyperparams)
```

```
Best hyerparameters:
        {'max_depth': 4,
         'max_features': 'log2',
         'min_samples_leaf': 0.1,
         'n_estimators': 400}
```

```
# Extract best model from 'grid_rf'
best_model = grid_rf.best_estimator_
# Predict the test set labels
y_pred = best_model.predict(X_test)
# Evaluate the test set RMSE
rmse_test = MSE(y_test, y_pred)**(1/2)
# Print the test set RMSE
print('Test set RMSE of rf: {:.2f}'.format(rmse_test))
```

```
Test set RMSE of rf: 3.89
```

```
# Define the dictionary 'params_rf'
params_rf = {
    'n_estimators': [100,350,500],
    'max_features': ['log2','auto','sqrt'],
    'min_samples_leaf': [2,10,30]
}
```

Set the RF hyperparameters

```
# Import GridSearchCV
from sklearn.model_selection import GridSearchCV
# Instantiate grid_rf
grid_rf = GridSearchCV(estimator=rf,
                       param_grid=params_rf,
                       scoring='neg_mean_squared_error',
                       cv=3,
                       verbose=1,
                       n_jobs=-1)
```

Search for the optimal forest

```
# Import mean_squared_error from sklearn.metrics as MSE
from sklearn.metrics import mean_squared_error as MSE
# Extract the best estimator
best_model = grid_rf.best_estimator_
# Predict test set labels
y_pred = best_model.predict(X_test)
# Compute rmse_test
rmse_test = MSE(y_test, y_pred)**(1/2)
# Print rmse_test
print('Test RMSE of best model: {:.3f}'.format(rmse_test))
```

Evaluate the optimal forest by finding the RMSE

Extreme Gradient Boosting with XGBoost

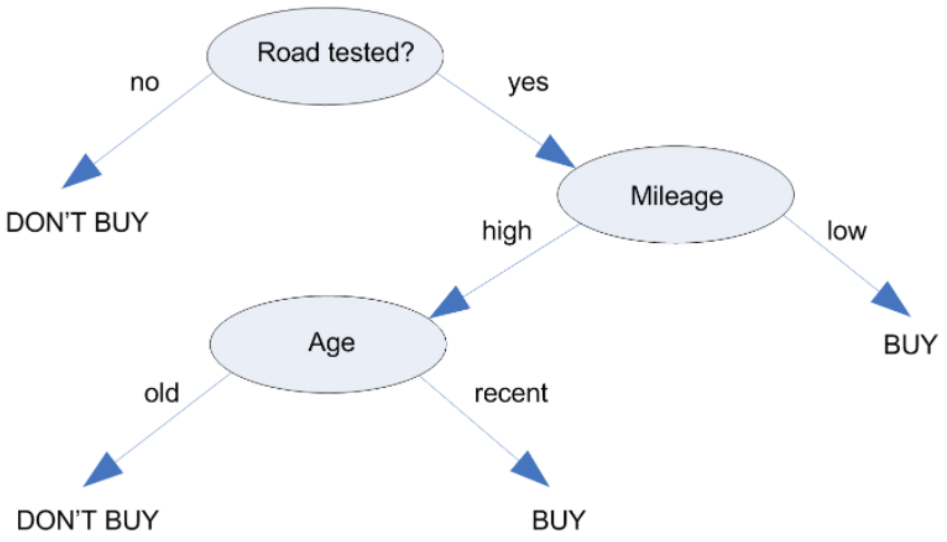
- High speed and performance
- Core algorithm is parallelizable
- Consistently outperforms single-algorithm methods
- State-of-the-art performance in many ML tasks

```
import xgboost as xgb
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
class_data = pd.read_csv("classification_data.csv")

X, y = class_data.iloc[:,:-1], class_data.iloc[:,-1]
X_train, X_test, y_train, y_test= train_test_split(X, y,
        test_size=0.2, random_state=123)
xg_cl = xgb.XGBClassifier(objective='binary:logistic',
        n_estimators=10, seed=123)
xg_cl.fit(X_train, y_train)

preds = xg_cl.predict(X_test)
accuracy = float(np.sum(preds==y_test))/y_test.shape[0]

print("accuracy: %f" % (accuracy))
```

```
accuracy: 0.78333
```

Example of using XGBoost

```
# Import xgboost
import xgboost as xgb
# Create arrays for the features and the target: X, y
X, y = churn_data.iloc[:,:-1], churn_data.iloc[:,-1]
# Create the training and test sets
X_train, X_test, y_train, y_test= train_test_split(X, y, test_size=0.2, random_state=123)
# Instantiate the XGBClassifier: xg_cl
xg_cl = xgb.XGBClassifier(objective='binary:logistic', n_estimators=10, seed=123)
# Fit the classifier to the training set
xg_cl.fit(X_train, y_train)
# Predict the labels of the test set: preds
preds = xg_cl.predict(X_test)
# Compute the accuracy: accuracy
accuracy = float(np.sum(preds==y_test))/y_test.shape[0]
print("accuracy: %f" % (accuracy))
```



Visualizing a decision tree

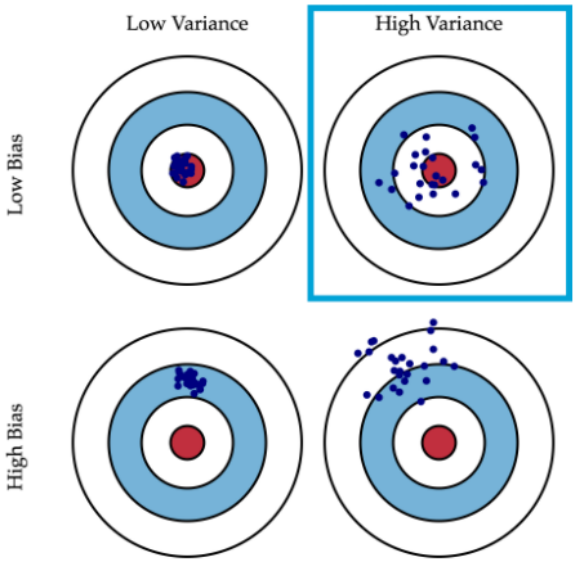
Decision trees as base learner

- Base learner → Individual learning algorithm in an ensemble algorithm
- Composed of a series of binary questions
- Predictions happen at the “leaves” of the tree

Decision trees and CART (Classification And Regression Tree)

- Constructed iteratively (one decision at a time) → Until a stopping criterion is met

Individual decision trees tend to overfit



Classification with XGBoost

CART → Each leaf **always** contains a real-valued score

→ Can later be converted into categories

```
# Import the necessary modules
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
# Create the training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=123)
# Instantiate the classifier: dt_clf_4
dt_clf_4 = DecisionTreeClassifier(max_depth=4)
# Fit the classifier to the training set
dt_clf_4.fit(X_train, y_train)
# Predict the labels of the test set: y_pred_4
y_pred_4 = dt_clf_4.predict(X_test)
# Compute the accuracy of the predictions: accuracy
accuracy = float(np.sum(y_pred_4==y_test))/y_test.shape[0]
print("accuracy:", accuracy)
```

Boosting overview

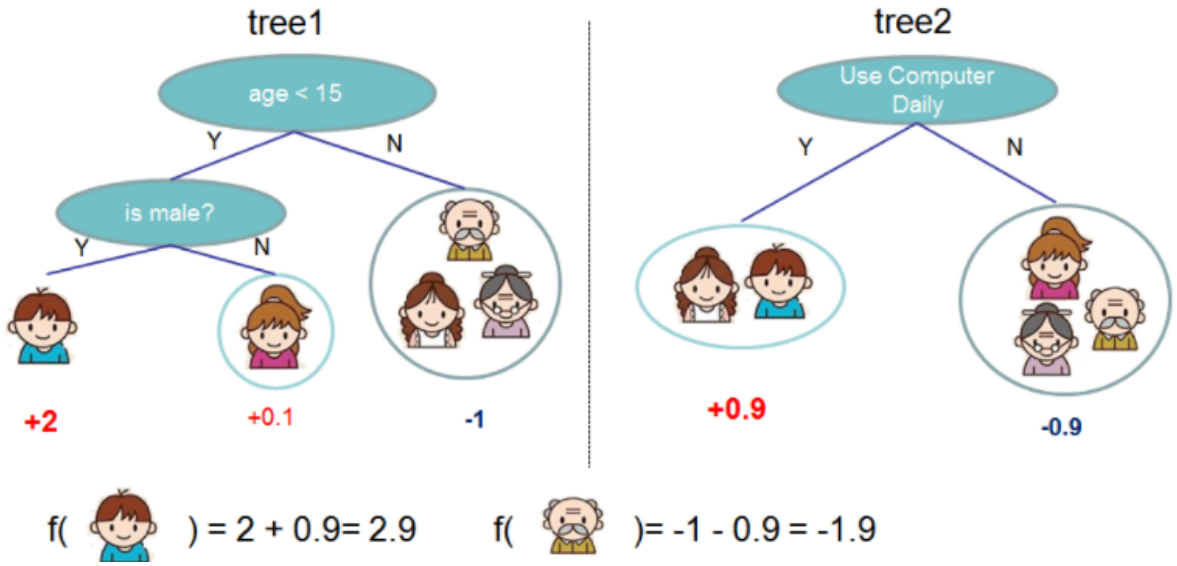
- Not a specific machine learning algorithm
- Concept that can be applied to a set of machine learning models “Meta-algorithm”
- Ensemble meta-algorithm used to convert many weak learner into a strong learner

Weak learners and strong learners

- Weak learner: ML algorithm that is slightly better than chance
  - Example: Decision tree whose predictions are slightly better than 50% (Better than flipping a coin for just a little!)
- Boosting converts a collection of weak learner into a strong learner
- Strong learner: Any algorithm that can be tuned to achieve good performance

How boosting is accomplished

- Iteratively learning a set of weak models on subsets of the data
- Weighing each weak prediction according to each weak learner’s performance
- Combine the weighted predictions to obtain a single weighted prediction
- … that is much better than the individual predictions themselves!



Model evaluation through cross-validation

- Cross-validation: Robust method for estimating the performance of a model on unseen data
- Generates many non-overlapping train/test splits on training data
- Reports the average test set performance across all data splits

```
import xgboost as xgb
import pandas as pd
churn_data = pd.read_csv("classification_data.csv")
churn_dmatrix = xgb.DMatrix(data=churn_data.iloc[:,:-1],
        label=churn_data.month_5_still_here)
params={"objective":"binary:logistic","max_depth":4}
cv_results = xgb.cv(dtrain=churn_dmatrix, params=params, nfold=4,
        num_boost_round=10, metrics="error", as_pandas=True)
print("Accuracy: %f" %((1-cv_results["test-error-mean"]).iloc[-1]))
```

```
Accuracy: 0.88315
```

Example: Cross-validation in XGBoost

```
# Create arrays for the features and the target: X, y
X, y = churn_data.iloc[:,:-1], churn_data.iloc[:,-1]
# Create the DMatrix from X and y: churn_dmatrix
churn_dmatrix = xgb.DMatrix(data=X, label=y)
# Create the parameter dictionary: params
params = {"objective":"reg:logistic", "max_depth":3}
# Perform cross-validation: cv_results
cv_results = xgb.cv(dtrain=churn_dmatrix, params=params,
                    nfold=3, num_boost_round=5,
                    metrics="error", as_pandas=True, seed=123)
# Print cv_results
print(cv_results)
# Print the accuracy
print(((1-cv_results["test-error-mean"]).iloc[-1]))
```

Measuring accuracy

Perform 3-fold cross-validation by calling xgb.cv(). dtrain is your churn_dmatrix, params is your parameter dictionary, nfold is the number of cross-validation folds (3), num_boost_round is the number of trees we want to build (5), metrics is the metric you want to compute (this will be "error", which we will convert to an accuracy).

	train-error-mean	train-error-std	test-error-mean	test-error-std
0	0.28232	0.002366	0.28378	0.001932
1	0.26951	0.001855	0.27190	0.001932
2	0.25605	0.003213	0.25798	0.003963
3	0.25090	0.001845	0.25434	0.003827
4	0.24654	0.001981	0.24852	0.000934

0.75148

```
# Perform cross_validation: cv_results
cv_results = xgb.cv(dtrain=churn_dmatrix, params=params,
                    nfold=3, num_boost_round=5,
                    metrics="auc", as_pandas=True, seed=123)
# Print cv_results
print(cv_results)
# Print the AUC
print((cv_results["test-auc-mean"]).iloc[-1])
```

Measuring AUC

	train-auc-mean	train-auc-std	test-auc-mean	test-auc-std
0	0.768893	0.001544	0.767863	0.002820
1	0.790864	0.006758	0.789157	0.006846
2	0.815872	0.003900	0.814476	0.005997
3	0.822959	0.002018	0.821682	0.003912
4	0.827528	0.000769	0.826191	0.001937

0.826191

Fantastic! An AUC of 0.84 is quite strong. As you have seen, XGBoost's learning API makes it very easy to compute any metric you may be interested in. In Chapter 3, you'll learn about techniques to fine-tune your XGBoost models to improve their performance even further. For now, it's time to learn a little about exactly when to use XGBoost.

When to use XGBoost → Supervised Learning

- Large number of training samples
    - Greater than 1000 training samples and less than 100 features
    - Number of features < number of training samples
- Mixture of categorical and numerical features
    - Or just numeric features

When to NOT use XGBoost

- Image recognition
- Computer vision
- NLP and understanding problems
- Number of training samples is significantly smaller than the number of features

Regression with XGBoost

Objective (loss) function

- Quantifies how far off a prediction is from the actual result
- Measures the difference between estimated and true values for some collection of data
- Goal: Find the model that yields the minimum value of the loss function

Common loss functions and XGBoost

- Loss function names in xgboost:
  - reg:linear → use for regression problems
  - reg:logistic → use for classification problems when you want just decision, not probability
  - binary:logistic → use when you want probability rather than just decision

Base Learners and why we need them

- XGBoost involves creating a meta-model that is composed of many individual models that combine to give a final prediction
- Individual models = base learners
- Want base learners that when combined create final prediction that is **non-linear**
- Each base learner should be good at distinguishing or predicting different parts of the dataset
- Two kinds of base learners: tree and linear

```
import xgboost as xgb
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

boston_data = pd.read_csv("boston_housing.csv")
X, y = boston_data.iloc[:,:-1],boston_data.iloc[:,-1]

X_train, X_test, y_train, y_test= train_test_split(X, y, test_size=0.2,
                                                   random_state=123)
xg_reg = xgb.XGBRegressor(objective='reg:linear', n_estimators=10,
                                                  seed=123)
xg_reg.fit(X_train, y_train)

preds = xg_reg.predict(X_test)
```

```
rmse = np.sqrt(mean_squared_error(y_test,preds))

print("RMSE: %f" % (rmse))
```

```
RMSE: 129043.2314
```

Trees as base learners example: Scikit-learn API

```
import xgboost as xgb
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

boston_data = pd.read_csv("boston_housing.csv")

X, y = boston_data.iloc[:,:-1],boston_data.iloc[:,-1]

X_train, X_test, y_train, y_test= train_test_split(X, y, test_size=0.2,
                                                   random_state=123)
DM_train = xgb.DMatrix(data=X_train,label=y_train)
DM_test =  xgb.DMatrix(data=X_test,label=y_test)
params = {"booster":"gblinear","objective":"reg:linear"}
xg_reg = xgb.train(params = params, dtrain=DM_train, num_boost_round=10)

preds = xg_reg.predict(DM_test)
```

```
rmse = np.sqrt(mean_squared_error(y_test,preds))

print("RMSE: %f" % (rmse))
```

```
RMSE: 124326.24465
```

Linear base learning: learning API only

```
# Create the training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=123)
# Instantiate the XGBRegressor: xg_reg
xg_reg = xgb.XGBRegressor(objective="reg:linear", n_estimators=10, seed=123)
# Note: You don't have to specify booster="gbtree" as this is the default.
# Fit the regressor to the training set
xg_reg.fit(X_train, y_train)
# Predict the labels of the test set: preds
preds = xg_reg.predict(X_test)
# Compute the rmse: rmse
rmse = np.sqrt(mean_squared_error(y_test, preds))
print("RMSE: %f" % (rmse))
```

Decision trees as base learners

[05:28:52] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
RMSE: 78847.401758

```
# Convert the training and testing sets into DMatrixes: DM_train, DM_test
DM_train = xgb.DMatrix(data=X_train, label=y_train)
DM_test =  xgb.DMatrix(data=X_test, label=y_test)
# Create the parameter dictionary: params
params = {"booster":"gblinear", "objective":"reg:linear"}
# Train the model: xg_reg
xg_reg = xgb.train(params = params, dtrain=DM_train, num_boost_round=5)
# Predict the labels of the test set: preds
preds = xg_reg.predict(DM_test)
# Compute and print the RMSE
rmse = np.sqrt(mean_squared_error(y_test,preds))
print("RMSE: %f" % (rmse))
```

Linear base learners

[05:32:43] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
RMSE: 43185.111949

```
# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)
# Create the parameter dictionary: params
params = {"objective":"reg:linear", "max_depth":4}
# Perform cross-validation: cv_results
cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=4, num_boost_round=5, metrics="mae",
as_pandas=True, seed=123)
# Print cv_results
print(cv_results)
# Extract and print final boosting round metric
print((cv_results["test-mae-mean"]).tail(1))
```

Test mae, if wanna change to “rmse” just replace 2 mae with rmse

Regularization in XGBoost

- Regularization is a control on model complexity
- Want models that are both accurate and as simple as possible
- Regularization parameters in XGBoost:
  - gamma → minimum loss reduction allowed for a split to occur
  - alpha → l1 regularization on leaf weights, larger values mean more regularization
  - lambda → regularization on leaf weights

```
import xgboost as xgb
import pandas as pd
boston_data = pd.read_csv("boston_data.csv")
X,y = boston_data.iloc[:,:-1],boston_data.iloc[:,-1]
boston_dmatrix = xgb.DMatrix(data=X,label=y)
params={"objective":"reg:linear","max_depth":4}
l1_params = [1,10,100]
rmses_l1=[]
for reg in l1_params:
    params["alpha"] = reg
    cv_results = xgb.cv(dtrain=boston_dmatrix, params=params,nfold=4,
                        num_boost_round=10,metrics="rmse",as_pandas=True,seed=123)
    rmses_l1.append(cv_results["test-rmse-mean"].tail(1).values[0])
print("Best rmse as a function of l1:")
print(pd.DataFrame(list(zip(l1_params,rmses_l1)), columns=["l1","rmse"]))
```

```
Best rmse as a function of l1:
    l1          rmse
0    1  69572.517742
1   10  73721.967141
2  100  82312.312413
```

Example: L1 regularization in XGBoost

Base learners in XGBoost:

- Linear base leaner:
  - Sum of linear terms
  - Boosted model is weighted sum of linear models (thus is itself linear)
  - Rarely used
- Tree base learner:
  - Decision tree

- Boosted model is weighted sum of decision trees (nonlinear)
- Almost exclusively used in XGBoost

```
pd.DataFrame(list(zip(list1,list2)),columns=
["list1","list2"]))
```

- `zip` creates a `generator` of parallel values:
  - `zip([1,2,3],["a","b""c"])` = `[1,"a"],[2,"b"],[3,"c"]`
  - `generators` need to be completely instantiated before they can be used in `DataFrame` objects
- `list()` instantiates the full generator and passing that into the `DataFrame` converts the whole expression

Creating DataFrames from multiple equal-length lists

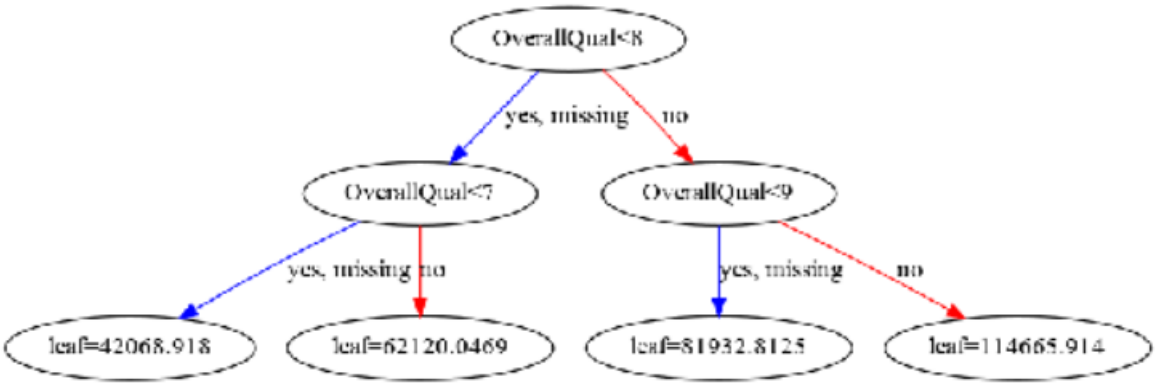
```
# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)
reg_params = [1, 10, 100]
# Create the initial parameter dictionary for varying l2 strength: params
params = {"objective":"reg:linear","max_depth":3}
# Create an empty list for storing rmses as a function of l2 complexity
rmses_l2 = []
# Iterate over reg_params
for reg in reg_params:
    # Update l2 strength
    params["lambda"] = reg
    # Pass this updated param dictionary into cv
    cv_results_rmse = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=2, num_boost_round=5,
metrics="rmse", as_pandas=True, seed=123)
    # Append best rmse (final round) to rmses_l2
    rmses_l2.append(cv_results_rmse["test-rmse-mean"].tail(1).values[0])
# Look at best rmse per l2 param
print("Best rmse as a function of l2:")
print(pd.DataFrame(list(zip(reg_params, rmses_l2)), columns=["l2", "rmse"]))
```

Using regularization in XGBoost

```
Best rmse as a function of l2:
    l2          rmse
0    1  52275.359375
1   10  57746.064453
2  100  76624.625000
```

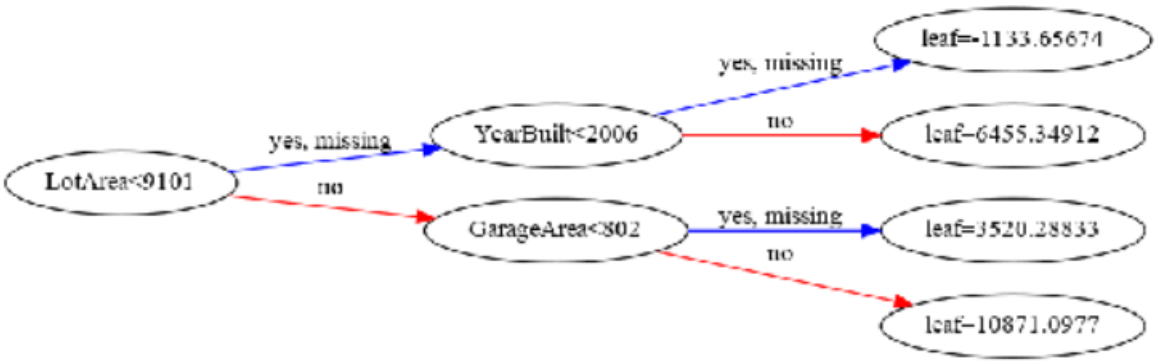
```
# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)
# Create the parameter dictionary: params
params = {"objective":"reg:linear", "max_depth":2}
# Train the model: xg_reg
xg_reg = xgb.train(params=params,
dtrain=housing_dmatrix, num_boost_round=10)
# Plot the first tree
xgb.plot_tree(xg_reg, num_trees=0)
plt.show()
# Plot the fifth tree
xgb.plot_tree(xg_reg, num_trees=4)
plt.show()
# Plot the last tree sideways
xgb.plot_tree(xg_reg, num_trees=9,rankdir="LR")
plt.show()
```

Visualizing individual XGBoost tree



First tree

Fifth tree

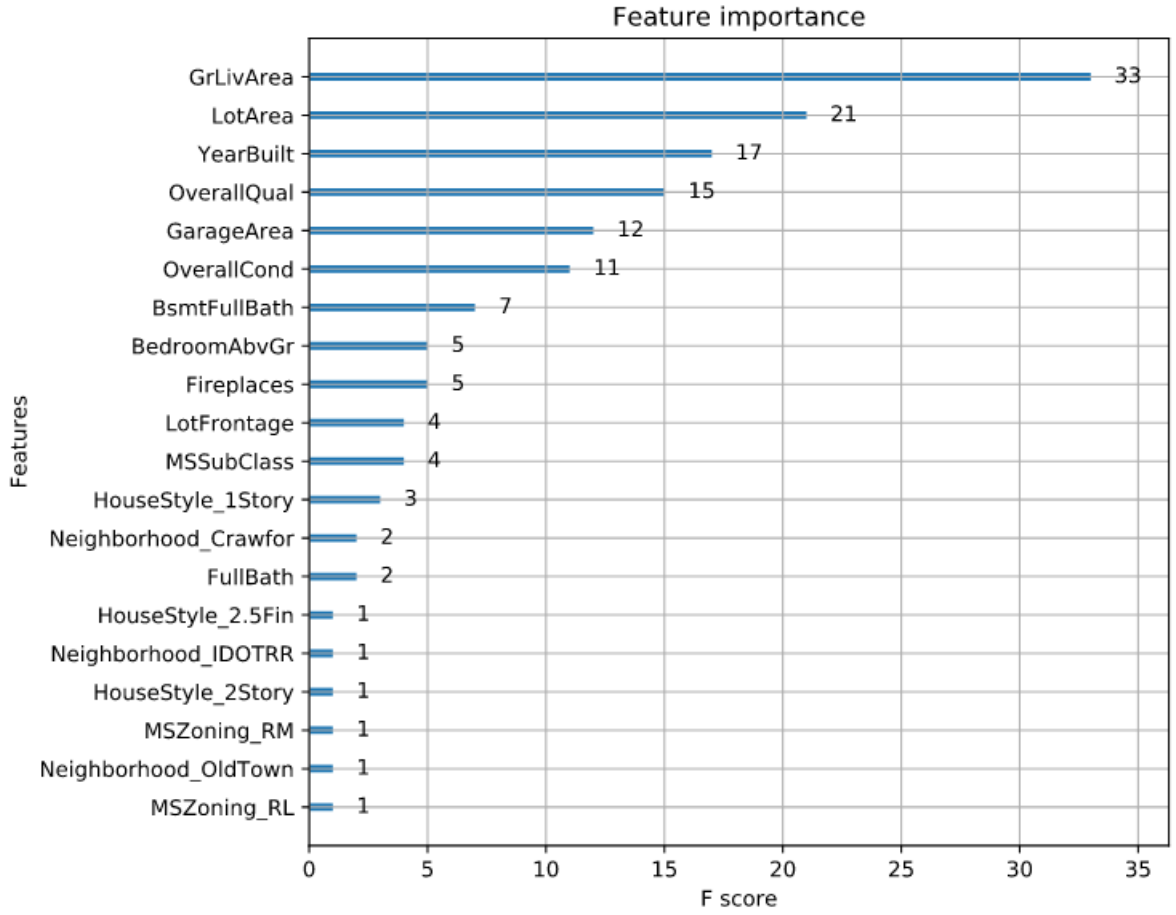


Last tree (10) in sideway [There are 10 trees]

They provide insight into how the model arrived at its final decisions and what splits it made to arrive at those decisions. This allows us to identify which features are the most important in determining house price. In the next exercise, you'll learn another way of visualizing feature importances.

```
# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)
# Create the parameter dictionary: params
params = {"objective":"reg:linear",
"max_depth":4}
# Train the model: xg_reg
xg_reg = xgb.train(params=params,
dtrain=housing_dmatrix, num_boost_round=10)
# Plot the feature importances
xgb.plot_importance(xg_reg)
plt.show()
```

Visualizing features importance



Fine tuning XGBoost model

Tuned model has lesser RMSE than untuned model (less error)

Untuned model example

```
import pandas as pd
import xgboost as xgb
import numpy as np
housing_data = pd.read_csv("ames_housing_trimmed_processed.csv")
X,y = housing_data[housing_data.columns.tolist()[:-1]],
        housing_data[housing_data.columns.tolist()[-1]]
housing_dmatrix = xgb.DMatrix(data=X,label=y)
untuned_params={"objective":"reg:linear"}
untuned_cv_results_rmse = xgb.cv(dtrain=housing_dmatrix,
        params=untuned_params,nfold=4,
        metrics="rmse",as_pandas=True,seed=123)
print("Untuned rmse: %f" %((untuned_cv_results_rmse["test-rmse-mean"]).tail(1)))
```

```
Untuned rmse: 34624.229980
```

Tuned model example

```
import pandas as pd
import xgboost as xgb
import numpy as np
housing_data = pd.read_csv("ames_housing_trimmed_processed.csv")
X,y = housing_data[housing_data.columns.tolist()[:-1]],
     housing_data[housing_data.columns.tolist()[-1]]
housing_dmatrix = xgb.DMatrix(data=X,label=y)
tuned_params = {"objective":"reg:linear",'colsample_bytree': 0.3,
    'learning_rate': 0.1, 'max_depth': 5}
tuned_cv_results_rmse = xgb.cv(dtrain=housing_dmatrix,
     params=tuned_params, nfold=4, num_boost_round=200, metrics="rmse",
     as_pandas=True, seed=123)
print("Tuned rmse: %f" %((tuned_cv_results_rmse["test-rmse-mean"]).tail(1)))
```

```
Tuned rmse: 29812.683594
```

```
# Create the DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)
# Create the parameter dictionary for each tree: params
params = {"objective":"reg:linear", "max_depth":3}
# Create list of number of boosting rounds
num_rounds = [5, 10, 15]
# Empty list to store final round rmse per XGBoost model
final_rmse_per_round = []
# Iterate over num_rounds and build one model per num_boost_round parameter
for curr_num_rounds in num_rounds:
    # Perform cross-validation: cv_results
    cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=3, num_boost_round=curr_num_rounds,
metrics="rmse", as_pandas=True, seed=123)
    # Append final round RMSE
    final_rmse_per_round.append(cv_results["test-rmse-mean"].tail().values[-1])
# Print the resultant DataFrame
num_rounds_rmses = list(zip(num_rounds, final_rmse_per_round))
print(pd.DataFrame(num_rounds_rmses,columns=["num_boosting_rounds","rmse"]))
```

Tuning the number of boosting rounds

	num_boosting_rounds	rmse
0	5	50903.298177
1	10	34774.191406
2	15	32895.098307

```
# Create your housing DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)
# Create the parameter dictionary for each tree: params
params = {"objective":"reg:linear", "max_depth":4}
# Perform cross-validation with early stopping: cv_results
cv_results = xgb.cv(params=params, dtrain=housing_dmatrix, nfold=3, num_boost_round=50,
early_stopping_rounds=10, metrics="rmse", as_pandas=True, seed=123)
# Print cv_results
print(cv_results)
```

Automated boosting round selection using early_stopping

Early stopping works by testing the XGBoost model after every boosting round against a hold-out dataset and stopping the creation of additional boosting rounds (thereby finishing training of the model early) if the hold-out metric ("rmse" in our case) does not improve for a given number of rounds. Here you will use the early_stopping_rounds parameter in xgb.cv() with a large possible number of boosting rounds (50). Bear in mind that if the holdout metric continuously improves up through when num_boost_rounds is reached, then early stopping does not occur.

Overview of XGBoost's hyperparameters

Common tree tunable parameters

- **learning rate:** learning rate/eta
- **gamma:** min loss reduction to create new tree split
- **lambda:** L2 reg on leaf weights
- **alpha:** L1 reg on leaf weights
- **max_depth:** max depth per tree
- **subsample:** % samples used per tree
- **colsample_bytree:** % features used per tree

Linear tunable parameters

- **lambda:** L2 reg on weights
- **alpha:** L1 reg on weights
- **lambda_bias:** L2 reg term on bias
- You can also tune the number of estimators used for both base model types!

```
# Create your housing DMatrix: housing_dmatrix
housing_dmatrix = xgb.DMatrix(data=X, label=y)
# Create the parameter dictionary for each tree (boosting round)
params = {"objective":"reg:linear", "max_depth":3}
# Create list of eta values and empty list to store final round rmse per xgboost model
eta_vals = [0.001, 0.01, 0.1]
best_rmse = []
# Systematically vary the eta
for curr_val in eta_vals:
    params["eta"] = curr_val
    # Perform cross-validation: cv_results
    cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=3, early_stopping_rounds=5,
num_boost_round=10, metrics="rmse", as_pandas=True, seed=123)
    # Append the final round rmse to best_rmse
    best_rmse.append(cv_results["test-rmse-mean"].tail().values[-1])
# Print the resultant DataFrame
print(pd.DataFrame(list(zip(eta_vals, best_rmse)), columns=["eta","best_rmse"]))
```

Tuning eta

```
# Create your housing DMatrix
housing_dmatrix = xgb.DMatrix(data=X,label=y)
# Create the parameter dictionary
params = {"objective":"reg:linear", "max_depth": 20}
# Create list of max_depth values
max_depths = [2, 5, 10, 20]
best_rmse = []
# Systematically vary the max_depth
for curr_val in max_depths:
    params["max_depth"] = curr_val
    # Perform cross-validation
    cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=2, early_stopping_rounds=5,
num_boost_round=10, metrics="rmse", as_pandas=True, seed=123)
    # Append the final round rmse to best_rmse
    best_rmse.append(cv_results["test-rmse-mean"].tail().values[-1])
# Print the resultant DataFrame
print(pd.DataFrame(list(zip(max_depths, best_rmse)),columns=["max_depth","best_rmse"]))
```

Tuning max_depth

	max_depth	best_rmse
0	2	37957.468750
1	5	35596.599610
2	10	36065.548829
3	20	36739.578125

```
# Create your housing DMatrix
housing_dmatrix = xgb.DMatrix(data=X,label=y)
# Create the parameter dictionary
params={"objective":"reg:linear","max_depth":3}
# Create list of hyperparameter values: colsample_bytree_vals
colsample_bytree_vals = [0.1, 0.5, 0.8, 1]
best_rmse = []
# Systematically vary the hyperparameter value
for curr_val in colsample_bytree_vals:
    params["colsample_bytree"] = curr_val
    # Perform cross-validation
    cv_results = xgb.cv(dtrain=housing_dmatrix, params=params, nfold=2,
                        num_boost_round=10, early_stopping_rounds=5,
                        metrics="rmse", as_pandas=True, seed=123)
    # Append the final round rmse to best_rmse
    best_rmse.append(cv_results["test-rmse-mean"].tail().values[-1])
# Print the resultant DataFrame
print(pd.DataFrame(list(zip(colsample_bytree_vals, best_rmse)), columns=["colsample_bytree","best_rmse"]))
```

Tuning colsample_bytree

	colsample_bytree	best_rmse
0	0.1	48193.451172
1	0.5	36013.544922
2	0.8	35932.962891
3	1.0	35836.042969

There are several other individual parameters that you can tune, such as "subsample", which dictates the fraction of the training data that is used during any given boosting round. Next up: Grid Search and Random Search to tune XGBoost hyperparameters more efficiently!

Review of grid search:

- Search exhaustively over a given set of hyperparameters, once per set of hyperparameters
- Number of models = number of distinct value per hyperparameter multiplied across each hyperparameter

```
import pandas as pd
import xgboost as xgb
import numpy as np
from sklearn.model_selection import GridSearchCV
housing_data = pd.read_csv("ames_housing_trimmed_processed.csv")
X, y = housing_data[housing_data.columns.tolist()[:-1]],
       housing_data[housing_data.columns.tolist()[-1]
housing_dmatrix = xgb.DMatrix(data=X,label=y)
gbm_param_grid = {'learning_rate': [0.01,0.1,0.5,0.9],
                  'n_estimators': [200],
                  'subsample': [0.3, 0.5, 0.9]}
gbm = xgb.XGBRegressor()
grid_mse = GridSearchCV(estimator=gbm,param_grid=gbm_param_grid,
            scoring='neg_mean_squared_error', cv=4, verbose=1)
grid_mse.fit(X, y)
print("Best parameters found: ",grid_mse.best_params_)
print("Lowest RMSE found: ", np.sqrt(np.abs(grid_mse.best_score_)))
```

```
Best parameters found: {'learning_rate': 0.1,
'n_estimators': 200, 'subsample': 0.5}
Lowest RMSE found:  28530.1829341
```

Grid search: example

Random search: review

- Create a (possibly infinite) range of hyperparameter values per hyperparameter that you would like to search over
- Set the number of iterations you would like for the random search to continue
- During each iteration, randomly draw a value in the range of specified values for each hyperparameter searched over and train/evaluate a model with those hyperparameters

```
import pandas as pd
import xgboost as xgb
import numpy as np
from sklearn.model_selection import RandomizedSearchCV
housing_data = pd.read_csv("ames_housing_trimmed_processed.csv")
X,y = housing_data[housing_data.columns.tolist()[:-1]],
      housing_data[housing_data.columns.tolist()[-1]]
housing_dmatrix = xgb.DMatrix(data=X,label=y)
gbm_param_grid = {'learning_rate': np.arange(0.05,1.05,.05),
                  'n_estimators': [200],
                  'subsample': np.arange(0.05,1.05,.05)}
gbm = xgb.XGBRegressor()
randomized_mse = RandomizedSearchCV(estimator=gbm, param_distributions=gbm_param_grid,
                        n_iter=25, scoring='neg_mean_squared_error', cv=4, verbose=1)
randomized_mse.fit(X, y)
print("Best parameters found: ",randomized_mse.best_params_)
print("Lowest RMSE found: ", np.sqrt(np.abs(randomized_mse.best_score_)))
```

```
Best parameters found: {'subsample': 0.60000000000000009,
'n_estimators': 200, 'learning_rate': 0.20000000000000001}
Lowest RMSE found: 28300.2374291
```

Random search: example

```
# Create the parameter grid: gbm_param_grid
gbm_param_grid = {
    'colsample_bytree': [0.3, 0.7],
    'n_estimators': [50],
    'max_depth': [2, 5]
}
# Instantiate the regressor: gbm
gbm = xgb.XGBRegressor()
# Perform grid search: grid_mse
grid_mse = GridSearchCV(estimator=gbm, param_grid=gbm_param_grid, cv=4, verbose=1,
scoring="neg_mean_squared_error")
# Fit grid_mse to the data
grid_mse.fit(X, y)
# Print the best parameters and lowest RMSE
print("Best parameters found: ", grid_mse.best_params_)
print("Lowest RMSE found: ", np.sqrt(np.abs(grid_mse.best_score_)))
```

Grid search with XGBoost

Best parameters found: {'colsample_bytree': 0.7, 'max_depth': 5, 'n_estimators': 50}
Lowest RMSE found: 29916.562522854438

```
# Create the parameter grid: gbm_param_grid
gbm_param_grid = {
    'n_estimators': [25],
    'max_depth': range(2, 12)
}
# Instantiate the regressor: gbm
gbm = xgb.XGBRegressor(n_estimators=10)
# Perform random search: grid_mse
randomized_mse = RandomizedSearchCV(estimator=gbm, param_distributions=gbm_param_grid, n_iter=5,
scoring="neg_mean_squared_error", cv=4, verbose=1)
# Fit randomized_mse to the data
randomized_mse.fit(X, y)
# Print the best parameters and lowest RMSE
print("Best parameters found: ", randomized_mse.best_params_)
print("Lowest RMSE found: ", np.sqrt(np.abs(randomized_mse.best_score_)))
```

Random search with XGBoost

```
Best parameters found:  {'n_estimators': 25, 'max_depth': 6}
   Lowest RMSE found:  36909.98213965752
```

Grid search and random search limitations

- Grid Search
  - Number of models you must build with every additional new parameter grows very quickly
- Random Search
  - Parameter space to explore can be massive
  - Randomly jumping throughout the space looking for a "best" result becomes a waiting game

Pipeline review

- Takes a list of named 2-tuples (name, pipeline_step) as input
- Tuples can contain any arbitrary scikit-learn compatible estimator or transformer object
- Pipeline implements fit/predict methods
- Can be used as input estimator into grid/randomized search and cross_val_score methods

```
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
names = ["crime","zone","industry","charles","no","rooms",
        "age", "distance","radial","tax","pupil","aam","lower","med_price"]

data = pd.read_csv("boston_housing.csv",names=names)

X, y = data.iloc[:,:-1], data.iloc[:,-1]
rf_pipeline = Pipeline[("st_scaler",
                StandardScaler()),
                ("rf_model",RandomForestRegressor())]

scores = cross_val_score(rf_pipeline,X,y,
scoring="neg_mean_squared_error",cv=10)
```

```
final_avg_rmse = np.mean(np.sqrt(np.abs(scores)))

print("Final RMSE:", final_avg_rmse)
```

```
Final RMSE: 4.54530686529
```

Scikit-learn pipeline example

Preprocessing I: LabelEncoder and OneHotEncoder

- `LabelEncoder` : Converts a categorical column of strings into integers
- `OneHotEncoder` : Takes the column of integers and encodes them as dummy variables
- Cannot be done within a pipeline

Preprocessing II: DictVectorizer

- Traditionally used in text processing
- Converts lists of feature mappings into vectors
- Need to convert DataFrame into a list of dictionary entries
- Explore the **scikit-learn documentation**

```
# Import LabelEncoder
from sklearn.preprocessing import LabelEncoder
# Fill missing values with 0
df.LotFrontage = df.LotFrontage.fillna(0)
# Create a boolean mask for categorical columns
categorical_mask = (df.dtypes == object)
# Get list of categorical column names
categorical_columns = df.columns[categorical_mask].tolist()
# Print the head of the categorical columns
print(df[categorical_columns].head())
# Create LabelEncoder object: le
le = LabelEncoder()
# Apply LabelEncoder to categorical columns
df[categorical_columns] = df[categorical_columns].apply(lambda x: le.fit_transform(x))
# Print the head of the LabelEncoded categorical columns
print(df[categorical_columns].head())
```

Encoding categorical columns I: LabelEncoder

```
# Import OneHotEncoder
from sklearn.preprocessing import OneHotEncoder
# Create OneHotEncoder: ohe
ohe = OneHotEncoder(categorical_features=categorical_mask, sparse=False)
# Apply OneHotEncoder to categorical columns - output is no longer a dataframe: df_encoded
df_encoded = ohe.fit_transform(df)
# Print first 5 rows of the resulting dataset - again, this will no longer be a pandas dataframe
print(df_encoded[:5, :])
# Print the shape of the original DataFrame
print(df.shape)
# Print the shape of the transformed array
print(df_encoded.shape)
```

Encoding categorical columns II: OneHotEncoder

```
# Import DictVectorizer
from sklearn.feature_extraction import DictVectorizer
# Convert df into a dictionary: df_dict
df_dict = df.to_dict("records")
# Create the DictVectorizer object: dv
dv = DictVectorizer(sparse=False)
# Apply dv on df: df_encoded
df_encoded = dv.fit_transform(df_dict)
# Print the resulting first five rows
print(df_encoded[:5,:])
# Print the vocabulary
print(dv.vocabulary_)
```

Encoding categorical columns III: DictVectorizer

```
# Import necessary modules
from sklearn.feature_extraction import DictVectorizer
from sklearn.pipeline import Pipeline
# Fill LotFrontage missing values with 0
X.LotFrontage = X.LotFrontage.fillna(0)
# Setup the pipeline steps: steps
steps = [("ohe_onestep", DictVectorizer(sparse=False)),
         ("xgb_model", xgb.XGBRegressor())]
# Create the pipeline: xgb_pipeline
xgb_pipeline = Pipeline(steps)
# Fit the pipeline
xgb_pipeline.fit(X.to_dict("records"), y)
```

Preprocessing within a pipeline

Scikit-learn pipeline example with XGBoost

```
import pandas as pd
import xgboost as xgb
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
names = ["crime","zone","industry","charles","no","rooms","age",
        "distance","radial","tax","pupil","aam","lower","med_price"]
data = pd.read_csv("boston_housing.csv",names=names)
X, y = data.iloc[:,:-1], data.iloc[:,-1]
xgb_pipeline = Pipeline[("st_scaler", StandardScaler()),
                        ("xgb_model",xgb.XGBRegressor())]
scores = cross_val_score(xgb_pipeline, X, y,
                         scoring="neg_mean_squared_error",cv=10)
final_avg_rmse = np.mean(np.sqrt(np.abs(scores)))
print("Final XGB RMSE:", final_avg_rmse)
```

```
Final RMSE: 4.02719593323
```

Additional components introduced for pipelines

- `sklearn_pandas` :
  - `DataFrameMapper` - Interoperability between `pandas` and `scikit-learn`
  - `CategoricalImputer` - Allow for imputation of categorical variables before conversion to integers
- `sklearn.preprocessing` :
  - `Imputer` - Native imputation of numerical columns in scikit-learn
- `sklearn.pipeline` :
  - `FeatureUnion` - combine multiple pipelines of features into a single pipeline of features

```
# Import necessary modules
from sklearn.feature_extraction import DictVectorizer
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
# Fill LotFrontage missing values with 0
X.LotFrontage = X.LotFrontage.fillna(0)
# Setup the pipeline steps: steps
steps = [("ohe_onestep", DictVectorizer(sparse=False)),
         ("xgb_model", xgb.XGBRegressor(max_depth=2, objective="reg:linear"))]
# Create the pipeline: xgb_pipeline
xgb_pipeline = Pipeline(steps)
# Cross-validate the model
cross_val_scores = cross_val_score(xgb_pipeline, X.to_dict("records"), y, cv=10,
scoring="neg_mean_squared_error")
# Print the 10-fold RMSE
print("10-fold RMSE: ", np.mean(np.sqrt(np.abs(cross_val_scores))))
```

Cross-validating your XGBoost model

https://archive.ics.uci.edu/ml/datasets/chronic_kidney_disease

You'll now continue your exploration of using pipelines with a dataset that requires significantly more wrangling. The chronic kidney disease dataset contains both categorical and numeric features, but contains lots of missing values. The goal here is to predict who has chronic kidney disease given various blood indicators as features.

As Sergey mentioned in the video, you'll be introduced to a new library, sklearn_pandas, that allows you to chain many more processing steps inside of a pipeline than are currently supported in scikit-learn. Specifically, you'll be able to impute missing categorical values directly using the Categorical_Imputer() class in sklearn_pandas, and the DataFrameMapper() class to apply any arbitrary sklearn-compatible transformer on DataFrame columns, where the resulting output can be either a NumPy array or DataFrame.

We've also created a transformer called a Dictifier that encapsulates converting a DataFrame using .to_dict("records") without you having to do it explicitly (and so that it works in a pipeline). Finally, we've also provided the list of feature names in kidney_feature_names, the target name in kidney_target_name, the features in X, and the target in y.

In this exercise, your task is to apply the CategoricalImputer to impute all of the categorical columns in the dataset. You can refer to how the numeric imputation mapper was created as a template. Notice the keyword arguments input_df=True and df_out=True? This is so that you can work with DataFrames instead of arrays. By default, the transformers are passed a numpy array of the selected columns as input, and as a result, the output of the DataFrame mapper is also an array. Scikit-learn transformers have historically been designed to work with numpy arrays, not pandas DataFrames, even though their basic indexing interfaces are similar.

```
# Import necessary modules
from sklearn_pandas import DataFrameMapper
from sklearn_pandas import CategoricalImputer
# Check number of nulls in each feature column
nulls_per_column = X.isnull().sum()
print(nulls_per_column)
# Create a boolean mask for categorical columns
categorical_feature_mask = X.dtypes == object
# Get list of categorical column names
categorical_columns = X.columns[categorical_feature_mask].tolist()
# Get list of non-categorical column names
non_categorical_columns = X.columns[~categorical_feature_mask].tolist()
# Apply numeric imputer
numeric_imputation_mapper = DataFrameMapper(
                                    [([numeric_feature], Imputer(strategy="median")) for numeric_feature in
non_categorical_columns],
                                    input_df=True,
                                    df_out=True
                                   )
# Apply categorical imputer
categorical_imputation_mapper = DataFrameMapper(
                                        [(category_feature, CategoricalImputer()) for category_feature in
categorical_columns],
                                        input_df=True,
                                        df_out=True
                                       )
```

Run Code Submit An

Kidney disease case study I: Categorical Imputer

```
# Import FeatureUnion
from sklearn.pipeline import FeatureUnion
# Combine the numeric and categorical transformations
numeric_categorical_union = FeatureUnion([
                                          ("num_mapper", numeric_imputation_mapper),
                                          ("cat_mapper", categorical_imputation_mapper)
                                         ])
```

Kidney disease case study II: Feature Union

```
# Create full pipeline
pipeline = Pipeline([
                     ("featureunion", numeric_categorical_union),
                     ("dictifier", Dictifier()),
                     ("vectorizer", DictVectorizer(sort=False)),
                     ("clf", xgb.XGBClassifier(max_depth=3))
                    ])
# Perform cross-validation
cross_val_scores = cross_val_score(pipeline, kidney_data, y, scoring="roc_auc", cv=3)
# Print avg. AUC
print("3-fold AUC: ", np.mean(cross_val_scores))
```

Kidney disease case study III: Full pipeline

<script.py> output:

3-fold AUC: 0.998637406769937

Tuning XGBoost hyperparameters in a pipeline

```
import pandas as pd
   ...: import xgboost as xgb
   ...: import numpy as np
   ...: from sklearn.preprocessing import StandardScaler
   ...: from sklearn.pipeline import Pipeline
   ...: from sklearn.model_selection import RandomizedSearchCV
names = ["crime","zone","industry","charles","no",
   ...: "rooms","age", "distance","radial","tax",
   ...: "pupil","aam","lower","med_price"]
data = pd.read_csv("boston_housing.csv",names=names)
X, y = data.iloc[:,:-1],data.iloc[:,-1]
xgb_pipeline = Pipeline[("st_scaler",
   ...: StandardScaler()), ("xgb_model",xgb.XGBRegressor())]
gbm_param_grid = {
   ...:     'xgb_model__subsample': np.arange(.05, 1, .05),
   ...:     'xgb_model__max_depth': np.arange(3,20,1),
   ...:     'xgb_model__colsample_bytree': np.arange(.1,1.05,.05) }
randomized_neg_mse = RandomizedSearchCV(estimator=xgb_pipeline,
   ...: param_distributions=gbm_param_grid, n_iter=10,
   ...: scoring='neg_mean_squared_error', cv=4)
randomized_neg_mse.fit(X, y)
```

Tuning XGBoost hyperparameters in a pipeline II

```
print("Best rmse: ", np.sqrt(np.abs(randomized_neg_mse.best_score_)))
```

```
Best rmse: 3.9966784203040677
```

```
print("Best model: ", randomized_neg_mse.best_estimator_)
```

```
Best model:  Pipeline(steps=[('st_scaler', StandardScaler(copy=True,
with_mean=True, with_std=True)),
('xgb_model', XGBRegressor(base_score=0.5, colsample_bylevel=1,
       colsample_bytree=0.95000000000000029, gamma=0, learning_rate=0.1,
       max_delta_step=0, max_depth=8, min_child_weight=1, missing=None,
       n_estimators=100, nthread=-1, objective='reg:linear', reg_alpha=0,
       reg_lambda=1, scale_pos_weight=1, seed=0, silent=True,
       subsample=0.90000000000000013))])
```

```
# Create the parameter grid
gbm_param_grid = {
    'clf__learning_rate': np.arange(0.05, 1, 0.05),
    'clf__max_depth': np.arange(3, 10, 1),
    'clf__n_estimators': np.arange(50, 200, 50)
}
# Perform RandomizedSearchCV
randomized_roc_auc = RandomizedSearchCV(pipeline, gbm_param_grid, cv=2, n_iter=2, scoring="roc_auc",
verbose=1)
# Fit the estimator
randomized_roc_auc.fit(X, y)
# Compute metrics
print("Best score: ", randomized_roc_auc.best_score_)
print("Best mode: ", randomized_roc_auc.best_estimator_)
```

```
Fitting 2 folds for each of 2 candidates, totalling 4 fits
  Best score:  0.9965333333333334
  Best mode:  Pipeline(memory=None,
        steps=[('featureunion',
              FeatureUnion(n_jobs=None,
                    transformer_list=[('num_mapper',
```

```
                DataFrameMapper(default=False,
                                df_out=True,
                                features=[(['age'],
                                           Imputer(axis=0,
                                                   copy=True,
                                                   missing_values='NaN',
                                                   strategy='median',
                                                   verbose=0)),
                                          (['bp'],
                                           Imputer(axis=0,
                                                   copy=True,
                                                   missing_values='NaN',
                                                   strategy='median',
                                                   verbose=0)),
                                          (['sg'],
                                           Imputer(axis=0,
                                                   copy=...
        XGBClassifier(base_score=0.5, booster='gbtree',
                      colsample_bylevel=1, colsample_bynode=1,
                      colsample_bytree=1, gamma=0,
                      learning_rate=0.9500000000000001,
                      max_delta_step=0, max_depth=4,
                      min_child_weight=1, missing=None,
                      n_estimators=100, n_jobs=1, nthread=None,
                      objective='binary:logistic', random_state=0,
                      reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
                      seed=None, silent=None, subsample=1,
                      verbosity=1))],
    verbose=False)
```

What We Have Not Covered (And How You Can Proceed)

- Using XGBoost for ranking/recommendation problems (Netflix/Amazon problem)
- Using more sophisticated hyperparameter tuning strategies for tuning XGBoost models (Bayesian Optimization)
- Using XGBoost as part of an ensemble of other models for regression/classification

Cluster Analysis in Python

What is a cluster

- A group of items with similar characteristics
- Google News: articles where similar words and word associations appear together
- Customer Segments

Clustering algorithms

- Hierarchical clustering
- K-means clustering
- Other clustering algorithms: DBSCAN, Gaussian methods

```
from scipy.cluster.hierarchy import linkage, fcluster
from matplotlib import pyplot as plt
import seaborn as sns, pandas as pd
```

```
x_coordinates = [80.1, 93.1, 86.6, 98.5, 86.4, 9.5, 15.2, 3.4,
                 10.4, 20.3, 44.2, 56.8, 49.2, 62.5, 44.0]
y_coordinates = [87.2, 96.1, 95.6, 92.4, 92.4, 57.7, 49.4,
                 47.3, 59.1, 55.5, 25.6, 2.1, 10.9, 24.1, 10.3]

df = pd.DataFrame({'x_coordinate': x_coordinates,
                   'y_coordinate': y_coordinates})
```

```
Z = linkage(df, 'ward')
df['cluster_labels'] = fcluster(Z, 3, criterion='maxclust')
```

```
sns.scatterplot(x='x_coordinate', y='y_coordinate',
                hue='cluster_labels', data = df)
plt.show()
```

Hierarchical clustering in SciPy

```
from scipy.cluster.vq import kmeans, vq
from matplotlib import pyplot as plt
import seaborn as sns, pandas as pd

import random
random.seed((1000,2000))
```

```
x_coordinates = [80.1, 93.1, 86.6, 98.5, 86.4, 9.5, 15.2, 3.4,
                 10.4, 20.3, 44.2, 56.8, 49.2, 62.5, 44.0]
y_coordinates = [87.2, 96.1, 95.6, 92.4, 92.4, 57.7, 49.4,
                 47.3, 59.1, 55.5, 25.6, 2.1, 10.9, 24.1, 10.3]

df = pd.DataFrame({'x_coordinate': x_coordinates, 'y_coordinate': y_coordinates})
```

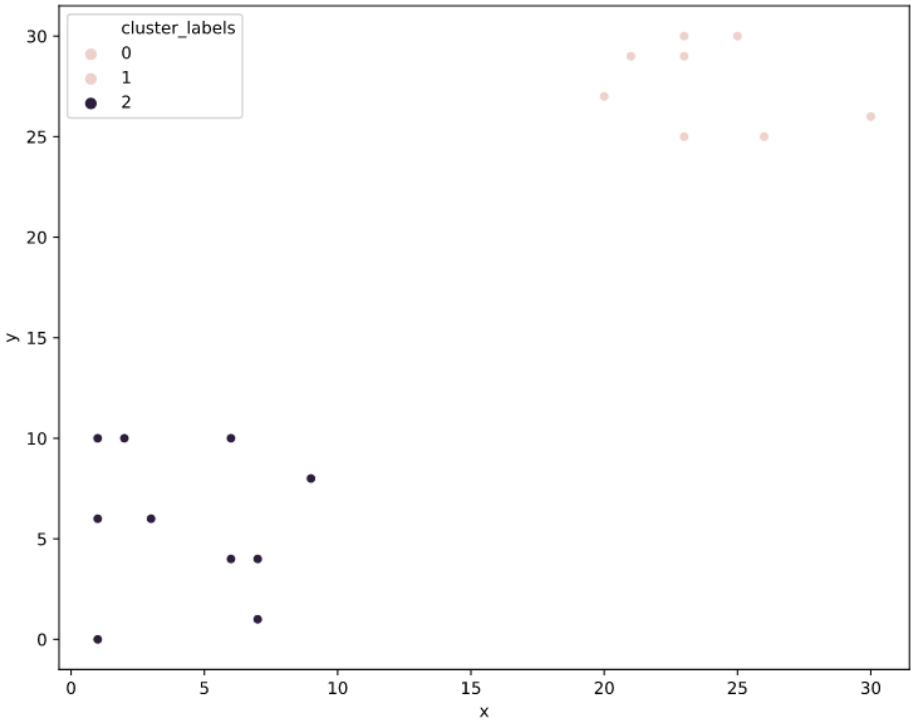
```
centroids,_ = kmeans(df, 3)
df['cluster_labels'], _ = vq(df, centroids)
```

```
sns.scatterplot(x='x_coordinate', y='y_coordinate',
                hue='cluster_labels', data = df)
plt.show()
```

K-means clustering in SciPy

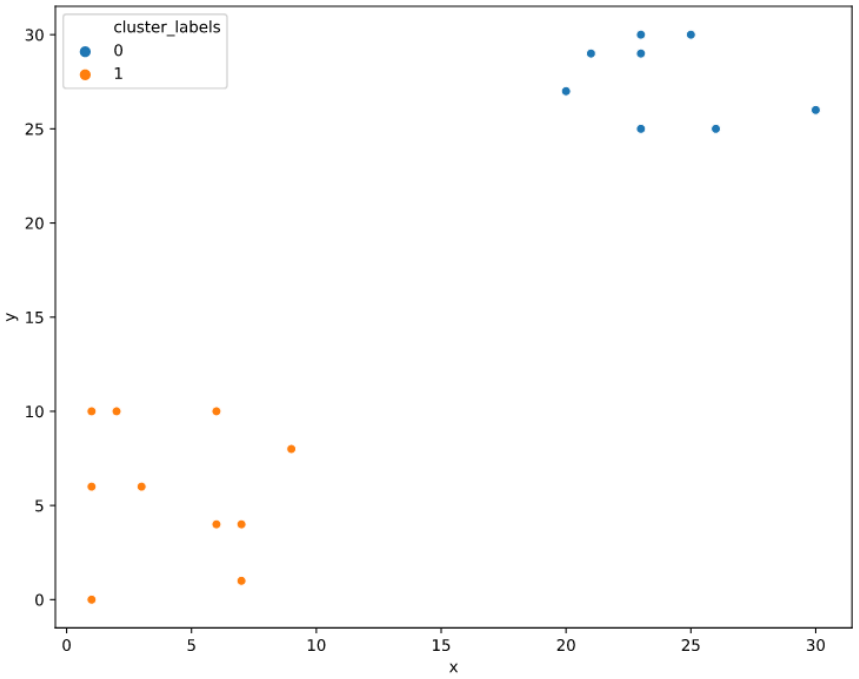
```
# Import linkage and fcluster functions
from scipy.cluster.hierarchy import linkage, fcluster
# Use the linkage() function to compute distance
Z = linkage(df, 'ward')
# Generate cluster labels
df['cluster_labels'] = fcluster(Z, 2, criterion='maxclust')
# Plot the points with seaborn
sns.scatterplot(x='x', y='y', hue='cluster_labels', data=df)
plt.show()
```

Pokémon sightings: hierarchical clustering



```
# Import kmeans and vq functions
from scipy.cluster.vq import kmeans, vq
# Compute cluster centers
centroids,_ = kmeans(df, 2)
# Assign cluster labels
df['cluster_labels'], _ = vq(df, centroids)
# Plot the points with seaborn
sns.scatterplot(x='x', y='y',
hue='cluster_labels', data=df)
plt.show()
```

Pokémon sightings: k-means clustering



Data preparation for cluster analysis

Why do we need to prepare data for clustering?

- Variables have incomparable units (product dimension in cm, price in $)
- Variables with same units have vastly different scale and variances (expenditures on cereals, travel)
- Data in raw form may lead to bias in clustering
- Clusters may be heavily dependent on one variable
- Solution: normalization of individual variables

Normalization: process of rescaling data to a standard deviation of 1

> x_new = x / std_dev(x)

```
from scipy.cluster.vq import whiten
```

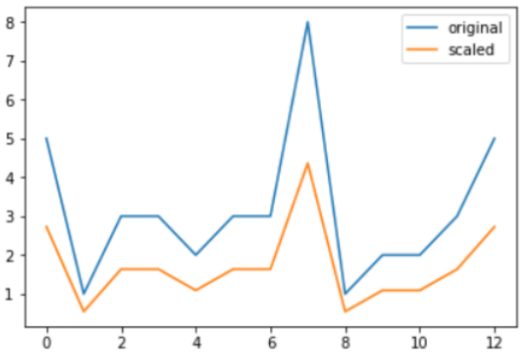
```
data = [5, 1, 3, 3, 2, 3, 3, 8, 1, 2, 2, 3, 5]
```

```
scaled_data = whiten(data)
print(scaled_data)
```

Illustration: normalization of data

```
# Import plotting library
from matplotlib import pyplot as plt

# Initialize original, scaled data
plt.plot(data,
         label="original")
plt.plot(scaled_data,
         label="scaled")
```

```
# Show legend and display plot
plt.legend()
plt.show()
```

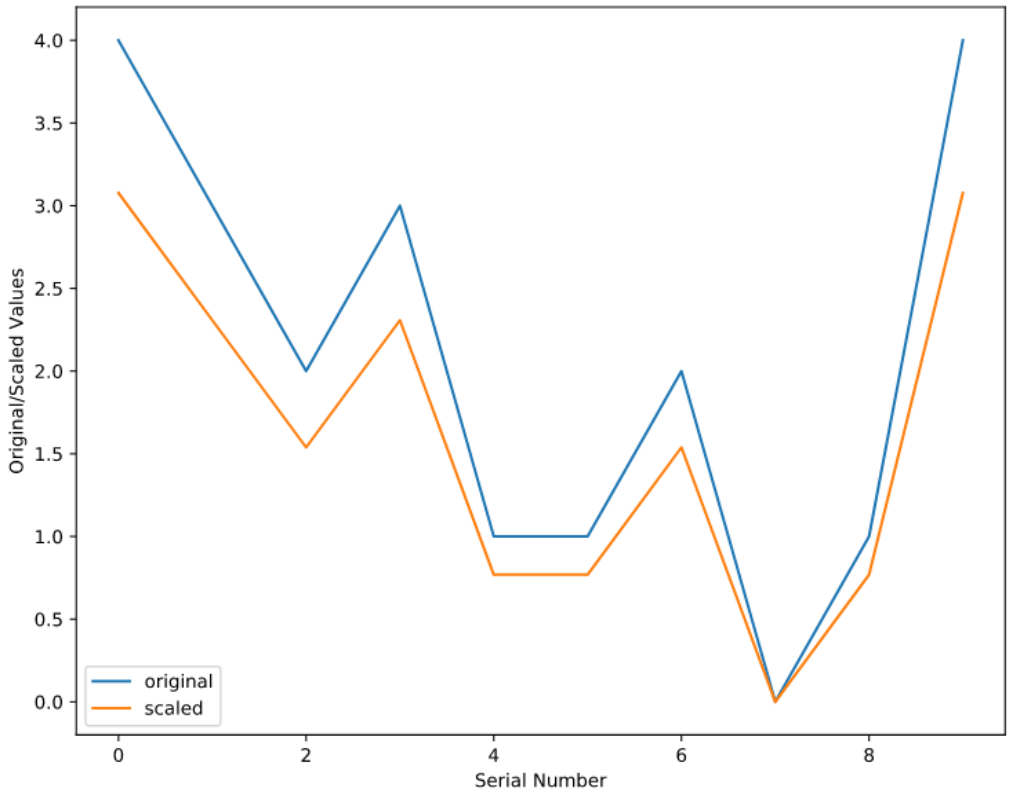



```
# Import the whiten function
from scipy.cluster.vq import whiten
goals_for = [4,3,2,3,1,1,2,0,1,4]
# Use the whiten() function to standardize the data
scaled_data = whiten(goals_for)
print(scaled_data)
```

Normalize basic list data

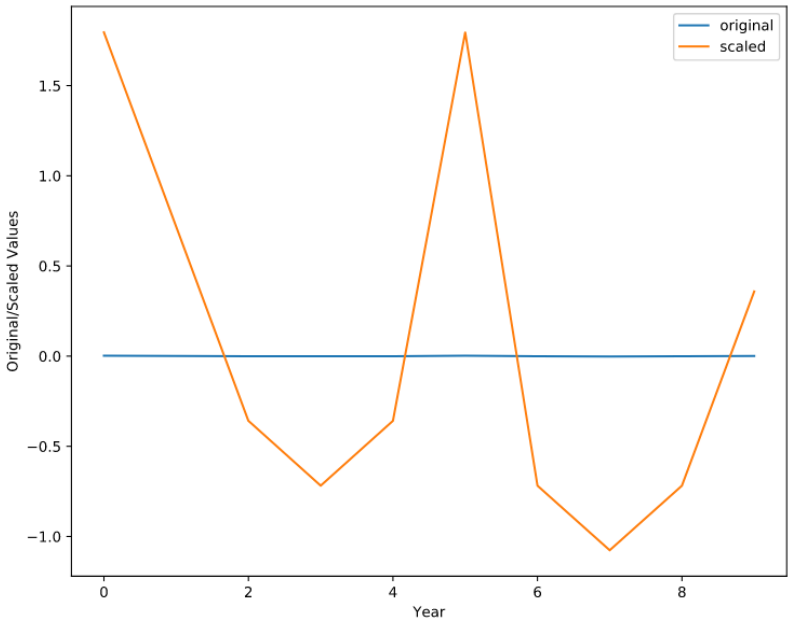
```
# Plot original data
plt.plot(goals_for, label='original')
# Plot scaled data
plt.plot(scaled_data, label='scaled')
# Show the legend in the plot
plt.legend()
# Display the plot
plt.show()
```

Visualize normalized data



```
# Prepare data
rate_cuts = [0.0025, 0.001, -0.0005, -0.001,
-0.0005, 0.0025, -0.001, -0.0015, -0.001, 0.0005]
# Use the whiten() function to standardize the
data
scaled_data = whiten(rate_cuts)
# Plot original data
plt.plot(rate_cuts, label='original')
# Plot scaled data
plt.plot(scaled_data, label='scaled')
plt.legend()
plt.show()
```

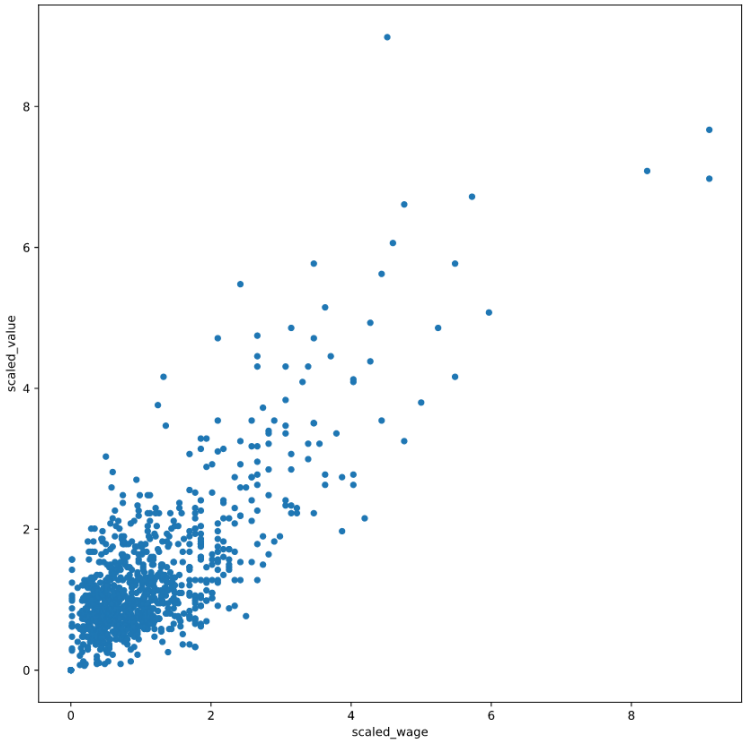
Normalization of small numbers




```
# Scale wage and value
fifa['scaled_wage'] = whiten(fifa['eur_wage'])
fifa['scaled_value'] = whiten(fifa['eur_value'])

# Plot the two columns in a scatter plot
fifa.plot(x='scaled_wage', y='scaled_value', kind = 'scatter')
plt.show()

# Check mean and standard deviation of scaled values
print(fifa[['scaled_wage', 'scaled_value']].describe())
```

FIFA 18: Normalize data

	scaled_wage	scaled_value
count	1000.00	1000.00
mean	1.12	1.31
std	1.00	1.00
min	0.00	0.00
25%	0.47	0.73
50%	0.85	1.02
75%	1.41	1.54
max	9.11	8.98



Basics of hierarchical clustering

Creating a distance matrix using linkage

```
scipy.cluster.hierarchy.linkage(observations,
                                method='single',
                                metric='euclidean',
                                optimal_ordering=False
)
```

- `method` : how to calculate the proximity of clusters
- `metric` : distance metric
- `optimal_ordering` : order data points

Which method should use?

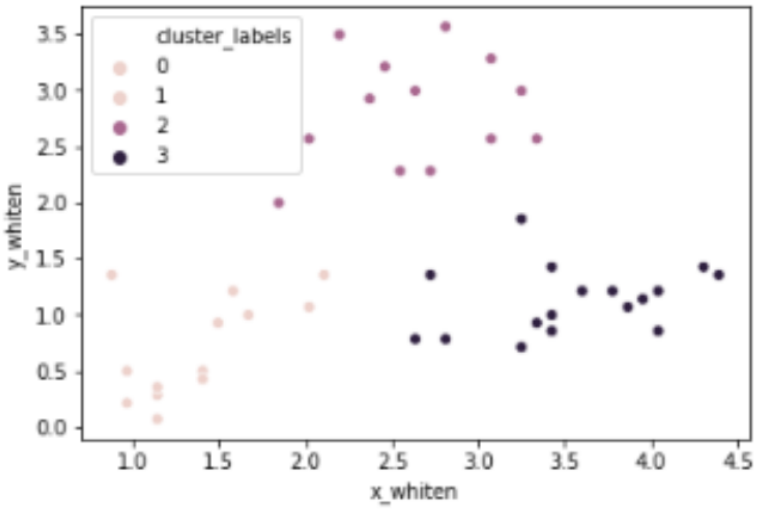
- single: based on two closest objects
- complete: based on two farthest objects
- average: based on the arithmetic mean of all objects
- centroid: based on the geometric mean of all objects
- median: based on the median of all objects
- ward: based on the sum of squares

Create cluster labels with fcluster

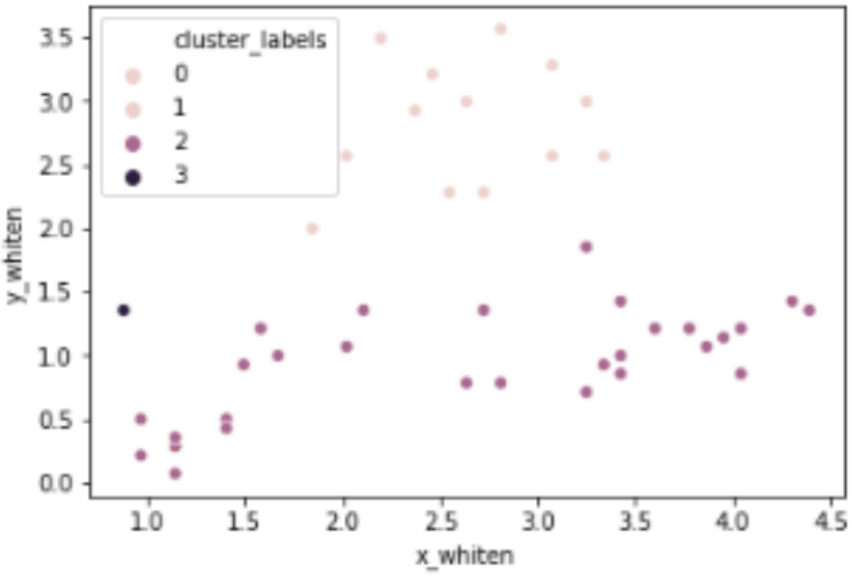
```
scipy.cluster.hierarchy.fcluster(distance_matrix,
                                 num_clusters,
                                 criterion
)
```

- `distance_matrix` : output of `linkage()` method
- `num_clusters` : number of clusters
- `criterion` : how to decide thresholds to form clusters

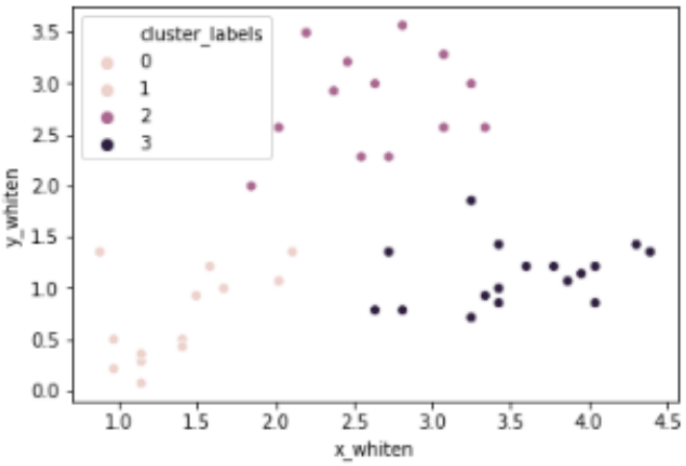
Hierarchical clustering with ward method



Hierarchical clustering with single method



Hierarchical clustering with complete method

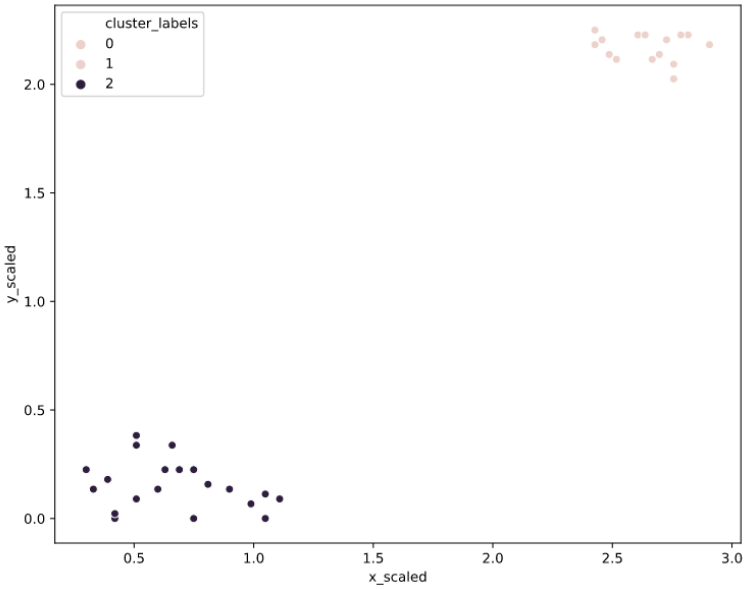


Final thoughts on selecting a method

- No one right method for all
- Need to carefully understand the distribution of data

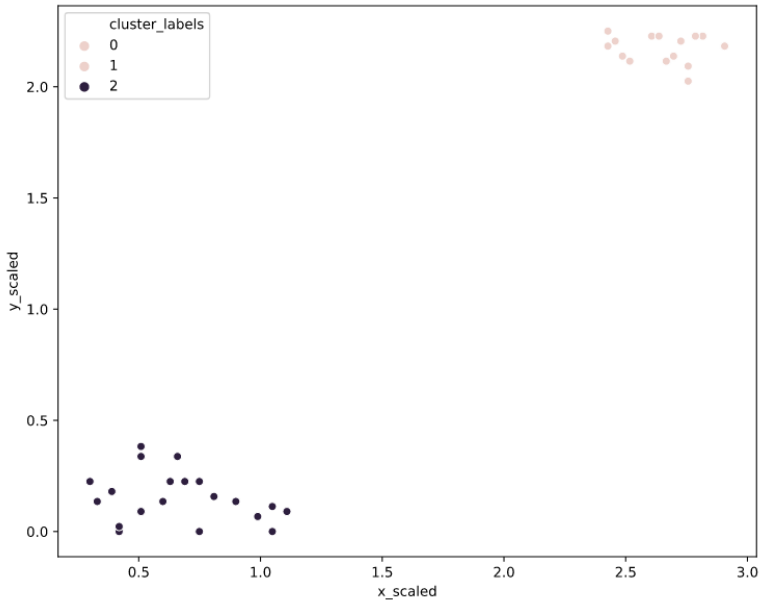
```
# Import the fcluster and linkage functions
from scipy.cluster.hierarchy import fcluster,
linkage

# Use the linkage() function
distance_matrix = linkage(comic_con[['x_scaled',
'y_scaled']], method = 'ward', metric =
'euclidean')

# Assign cluster labels
comic_con['cluster_labels'] = fcluster
(distance_matrix, 2, criterion='maxclust')

# Plot clusters
sns.scatterplot(x='x_scaled', y='y_scaled',
                hue='cluster_labels', data =
comic_con)
plt.show()
```

Hierarchical clustering: ward method



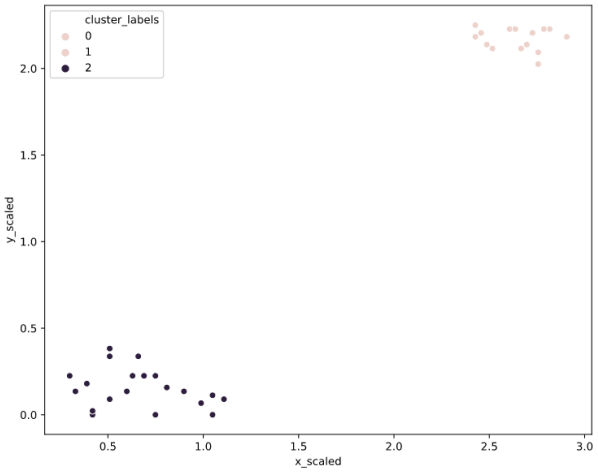
```
# Import the fcluster and linkage functions
from scipy.cluster.hierarchy import fcluster,
linkage
# Use the linkage() function
distance_matrix = linkage(comic_con[['x_scaled',
'y_scaled']], method = 'single', metric =
'euclidean')
# Assign cluster labels
comic_con['cluster_labels'] = fcluster
(distance_matrix, 2, criterion='maxclust')
# Plot clusters
sns.scatterplot(x='x_scaled', y='y_scaled',
                hue='cluster_labels', data =
comic_con)
plt.show()
```

Hierarchical clustering: single method




```
# Import the fcluster and linkage functions
from scipy.cluster.hierarchy import fcluster,
linkage
# Use the linkage() function
distance_matrix = linkage(comic_con[['x_scaled',
'y_scaled']], method='complete',
metric='euclidean')
# Assign cluster labels
comic_con['cluster_labels'] = fcluster
(distance_matrix, 2, criterion='maxclust')
# Plot clusters
sns.scatterplot(x='x_scaled', y='y_scaled',
                hue='cluster_labels', data =
comic_con)
plt.show()
```

Hierarchical clustering: complete method



Visualize clusters

An introduction to seaborn

- `seaborn` : a Python data visualization library based on `matplotlib`
- Has better, easily modifiable aesthetics than matplotlib!
- Contains functions that make data visualization tasks easy in the context of data analytics
- Use case for clustering: `hue` parameter for plots

Visualize clusters with matplotlib

```
from matplotlib import pyplot as plt
```

```
df = pd.DataFrame({'x': [2, 3, 5, 6, 2],
                   'y': [1, 1, 5, 5, 2],
                   'labels': ['A', 'A', 'B', 'B', 'A']})
colors = {'A':'red', 'B':'blue'}
df.plot.scatter(x='x',
                y='y',
                c=df['labels'].apply(lambda x: colors[x]))
plt.show()
```

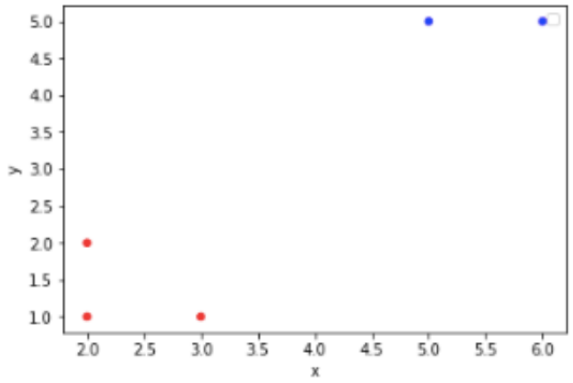
Visualize clusters with seaborn

```
from matplotlib import pyplot as plt
import seaborn as sns
```

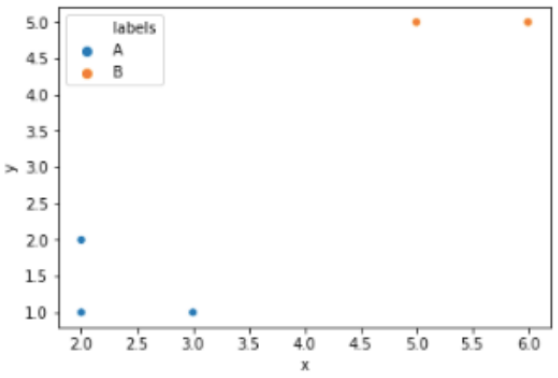
```
df = pd.DataFrame({'x': [2, 3, 5, 6, 2],
                   'y': [1, 1, 5, 5, 2],
                   'labels': ['A', 'A', 'B', 'B', 'A']})
sns.scatterplot(x='x',
                y='y',
                hue='labels',
                data=df)
plt.show()
```

Comparison of both methods of visualization

MATPLOTLIB PLOT

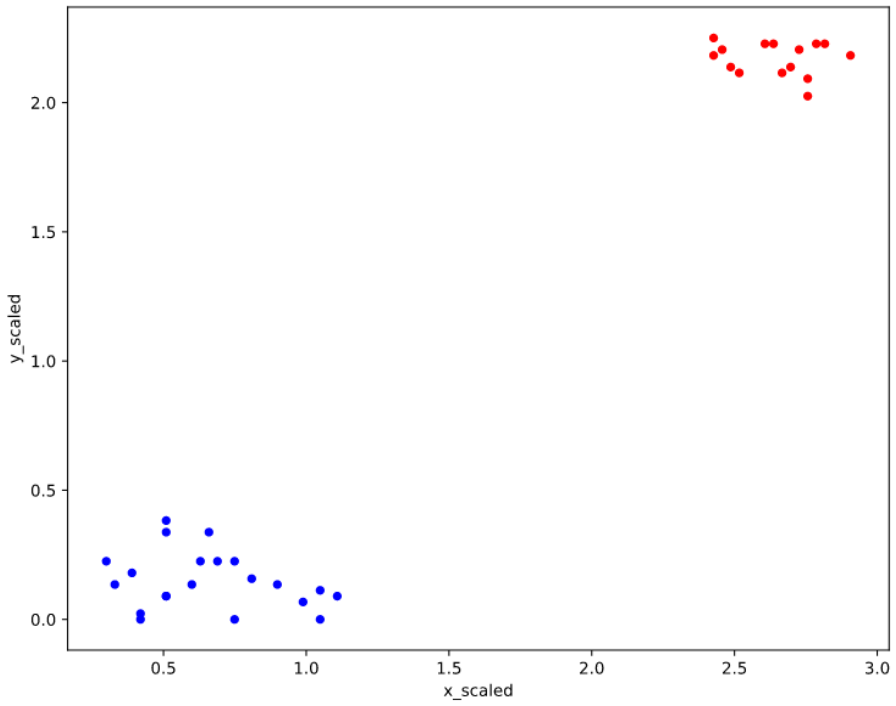


SEABORN PLOT



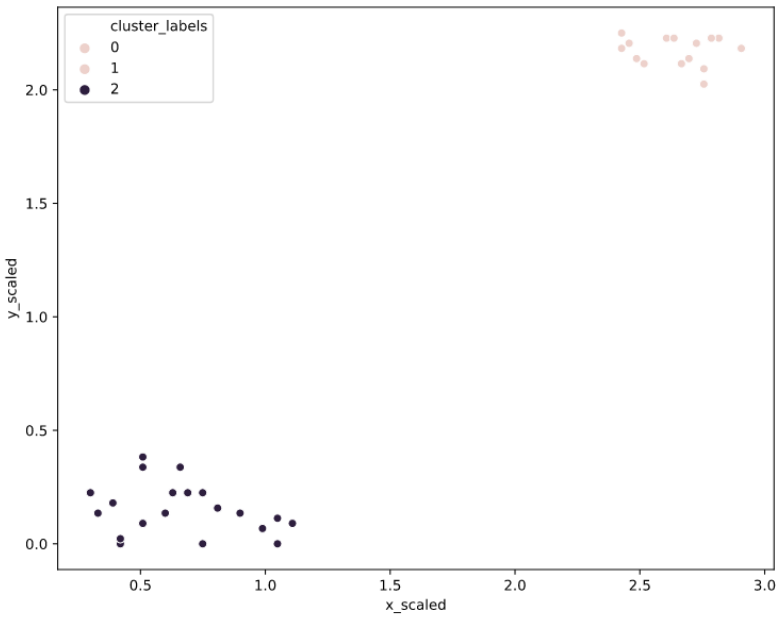

```
# Import the pyplot class
import matplotlib.pyplot as plt
# Define a colors dictionary for clusters
colors = {1:'red', 2:'blue'}
# Plot a scatter plot
comic_con.plot.scatter(x='x_scaled',
                       y='y_scaled',
                       c=comic_con
['cluster_labels'].apply(lambda c: colors[c]))
plt.show()
```

Visualize clusters with matplotlib



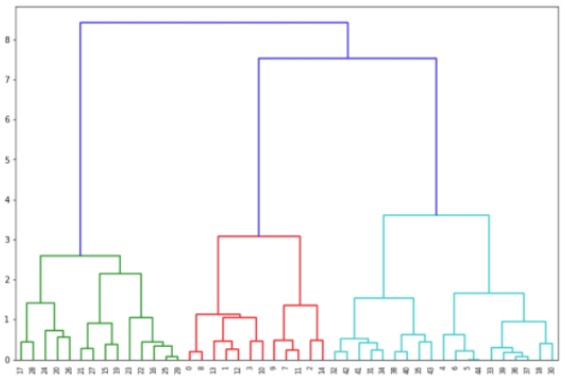
```
# Import the seaborn module
import matplotlib.pyplot as plt
import seaborn as sns
# Plot a scatter plot using seaborn
sns.scatterplot(x='x_scaled',
                y='y_scaled',
                hue='cluster_labels',
                data = comic_con)
plt.show()
```

Visualize clusters with seaborn



Introduction to dendrograms

- Strategy till now - decide clusters on visual inspection
- Dendrograms help in showing progressions as clusters are merged
- A dendrogram is a branching diagram that demonstrates how each cluster is composed by branching out into its child nodes

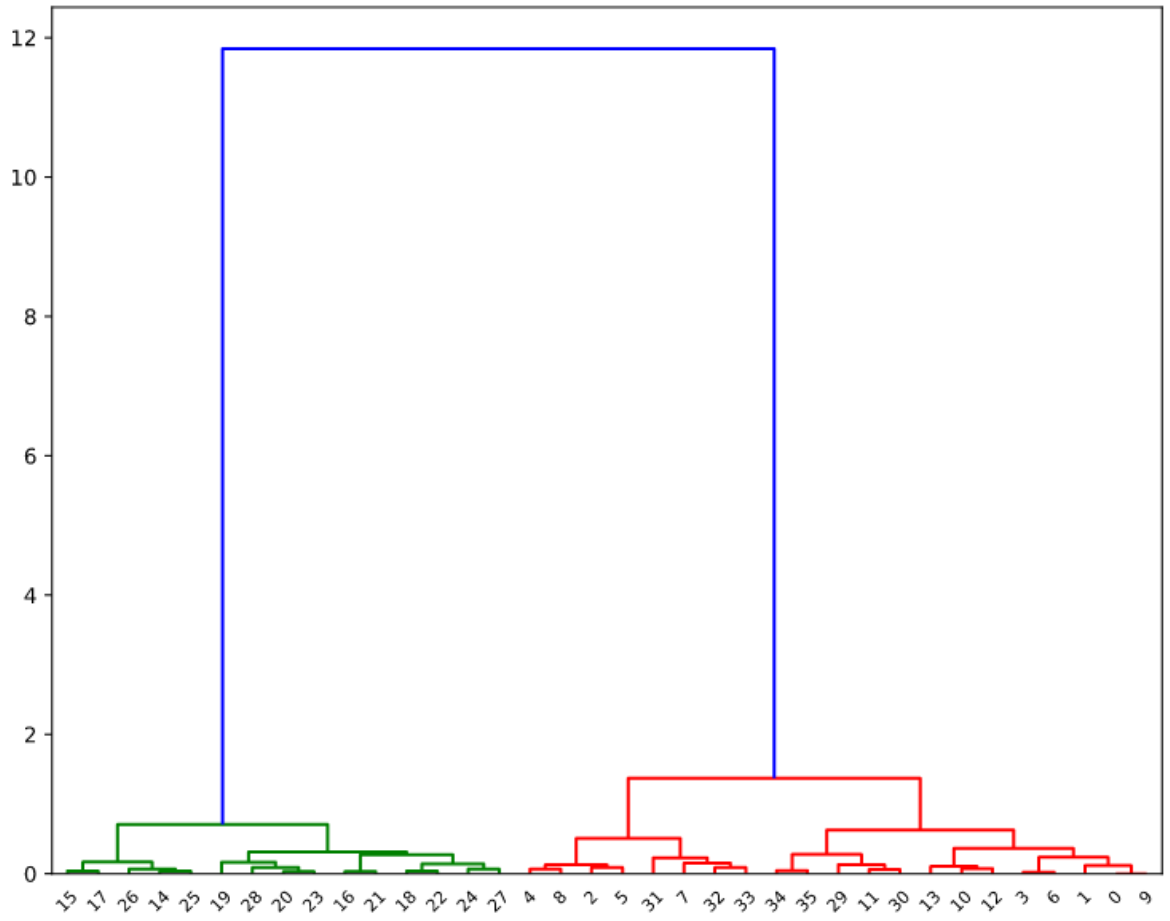


Create a dendrogram in SciPy

```
from scipy.cluster.hierarchy import dendrogram
```

```
Z = linkage(df[['x_whiten', 'y_whiten']],
            method='ward',
            metric='euclidean')
dn = dendrogram(Z)
plt.show()
```

```
# Import the dendrogram function
from scipy.cluster.hierarchy import dendrogram
Z = linkage(comic_con[['x_scaled','y_scaled']],
method='ward',metric='euclidean')
# Create a dendrogram
dn = dendrogram(Z)
# Display the dendogram
plt.show()
```



Measuring speed in hierarchical clustering

- `timeit` module
- Measure the speed of `.linkage()` method
- Use randomly generated points
- Run various iterations to extrapolate

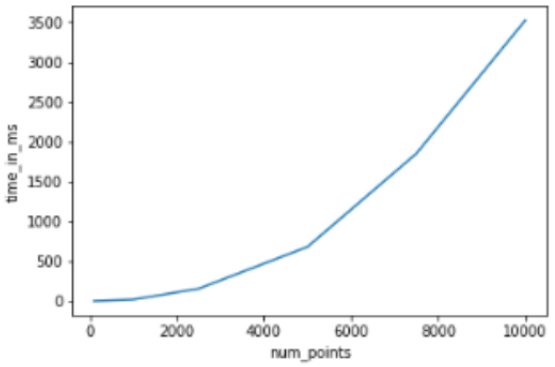
Use of timeit module

```
from scipy.cluster.hierarchy import linkage
import pandas as pd
import random, timeit
points = 100
df = pd.DataFrame({'x': random.sample(range(0, points), points),
                   'y': random.sample(range(0, points), points)})
%timeit linkage(df[['x', 'y']], method = 'ward', metric = 'euclidean')
```

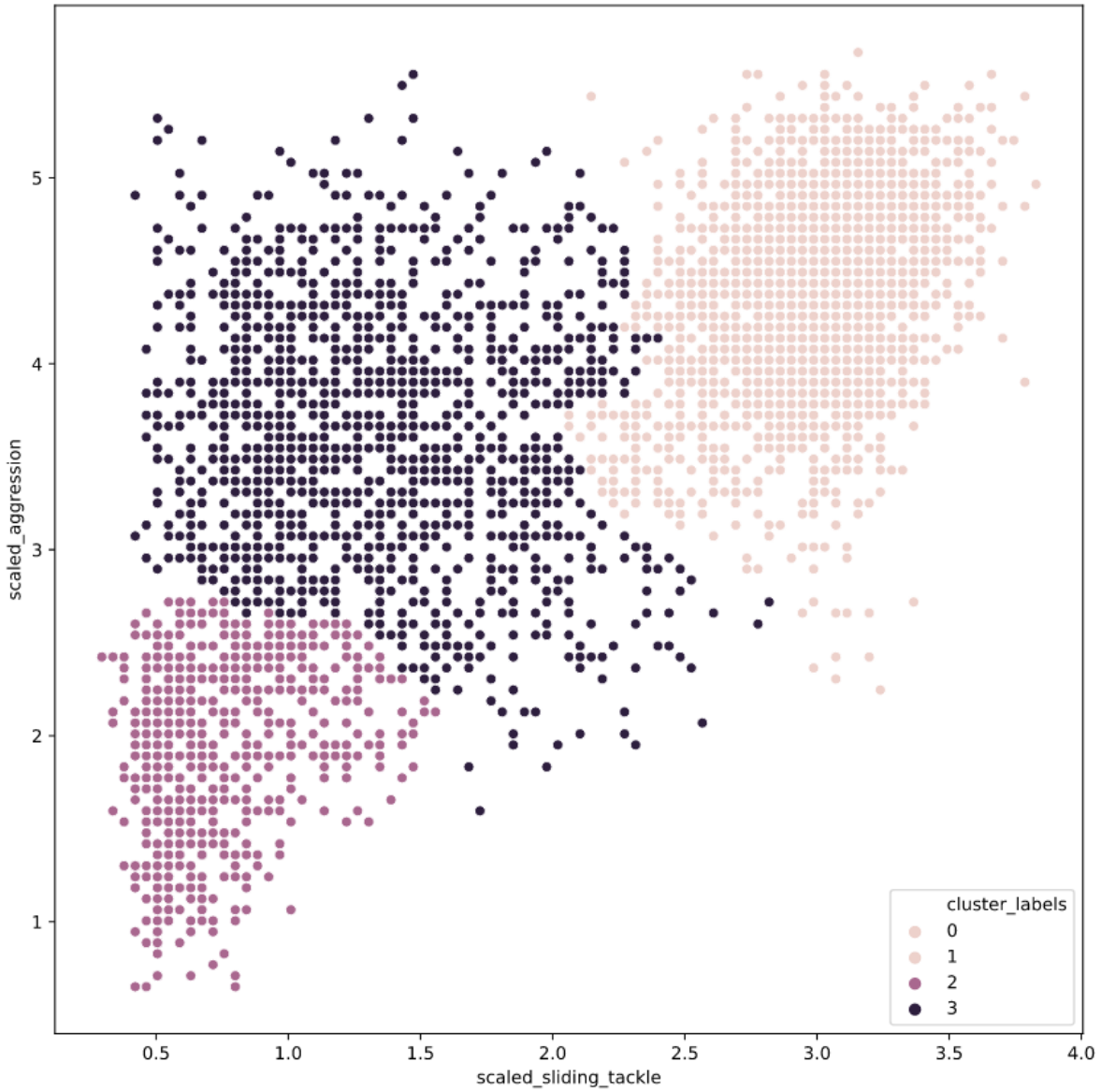
```
1.02 ms ± 133 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Comparison of runtime of linkage method

- Increasing runtime with data points
- Quadratic increase of runtime
- Not feasible for large datasets




```
# Fit the data into a hierarchical clustering
algorithm
distance_matrix = linkage(fifa[
['scaled_sliding_tackle', 'scaled_aggression']],
'ward')
# Assign cluster labels to each row of data
fifa['cluster_labels'] = fcluster(distance_matrix, 3,
criterion='maxclust')
# Display cluster centers of each cluster
print(fifa[['scaled_sliding_tackle',
'scaled_aggression', 'cluster_labels']].groupby
('cluster_labels').mean())
# Create a scatter plot through seaborn
sns.scatterplot(x='scaled_sliding_tackle',
y='scaled_aggression', hue='cluster_labels',
data=fifa)
plt.show()
```



K-means clustering

- Critical drawback of hierarchical clustering: runtime
- K-means run significantly faster on large datasets

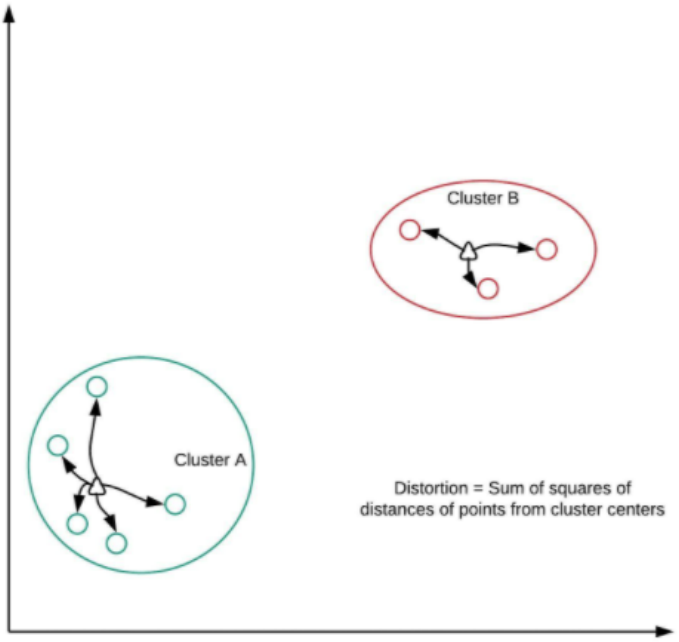
Step 1: Generate cluster centers

```
kmeans(obs, k_or_guess, iter, thresh, check_finite)
```

- `obs` : standardized observations
- `k_or_guess` : number of clusters
- `iter` : number of iterations (default: 20)
- `thres` : threshold (default: 1e-05)
- `check_finite` : whether to check if observations contain only finite numbers (default: True)

Returns two objects: cluster centers, distortion

How is distortion calculated?



Step 2: Generate cluster labels

```
vq(obs, code_book, check_finite=True)
```

- `obs` : standardized observations
- `code_book` : cluster centers
- `check_finite` : whether to check if observations contain only finite numbers (default: True)

Returns two objects: a list of cluster labels, a list of distortions

A note on distortions

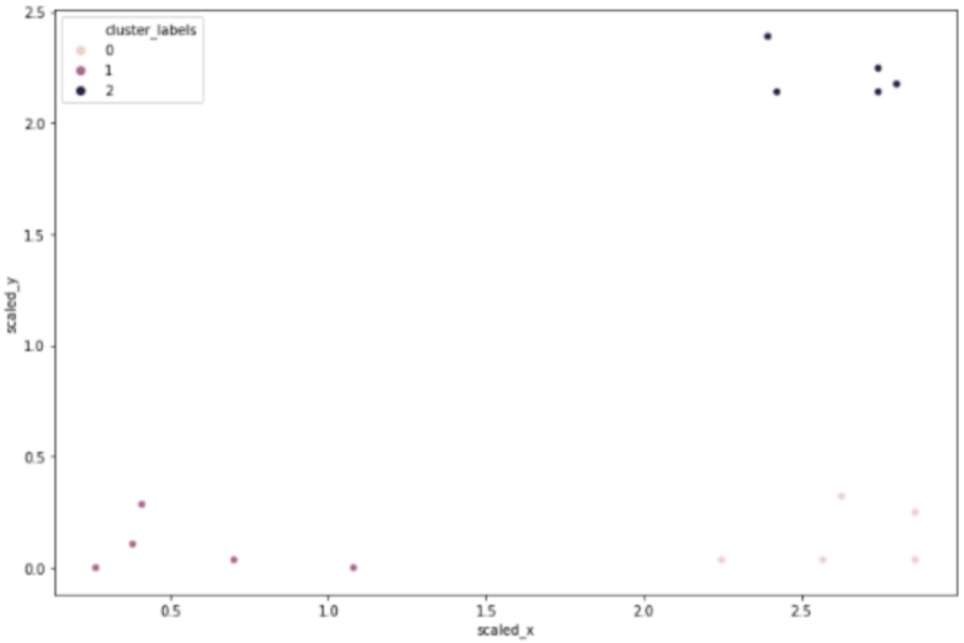
- `kmeans` returns a single value of distortions
- `vq` returns a list of distortions.

Running k-means

```
# Import kmeans and vq functions
from scipy.cluster.vq import kmeans, vq
```

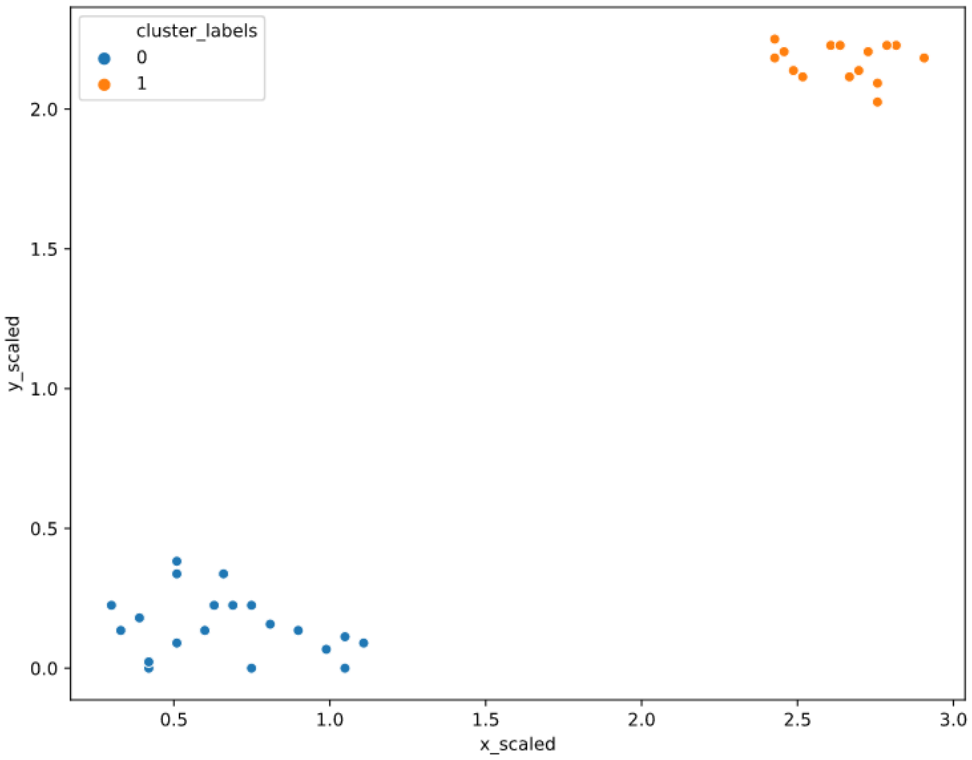
```
# Generate cluster centers and labels
cluster_centers, _ = kmeans(df[['scaled_x', 'scaled_y']], 3)
df['cluster_labels'], _ = vq(df[['scaled_x', 'scaled_y']], cluster_centers)
```

```
# Plot clusters
sns.scatterplot(x='scaled_x', y='scaled_y', hue='cluster_labels', data=df)
plt.show()
```



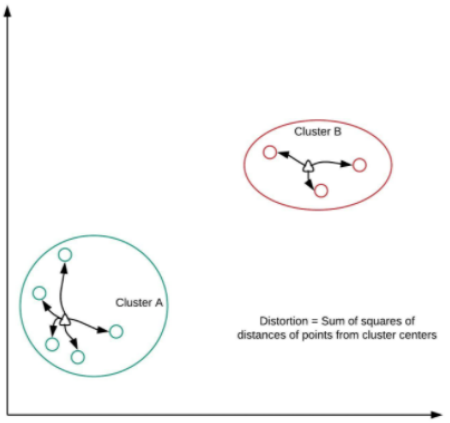

```
# Import the kmeans and vq functions
from scipy.cluster.vq import kmeans, vq
# Generate cluster centers
cluster_centers, distortion = kmeans(comic_con[
['x_scaled','y_scaled']], 2)
# Assign cluster labels
comic_con['cluster_labels'], distortion_list = vq
(comic_con[['x_scaled','y_scaled']], cluster_centers)
# Plot clusters
sns.scatterplot(x='x_scaled', y='y_scaled',
                hue='cluster_labels', data = comic_con)
plt.show()
```

K-means clustering: first exercise



Distortions revisited

- Distortion: sum of squared distances of points from cluster centers
- Decreases with an increasing number of clusters
- Becomes zero when the number of clusters equals the number of points
- Elbow plot: line plot between cluster centers and distortion



Elbow method

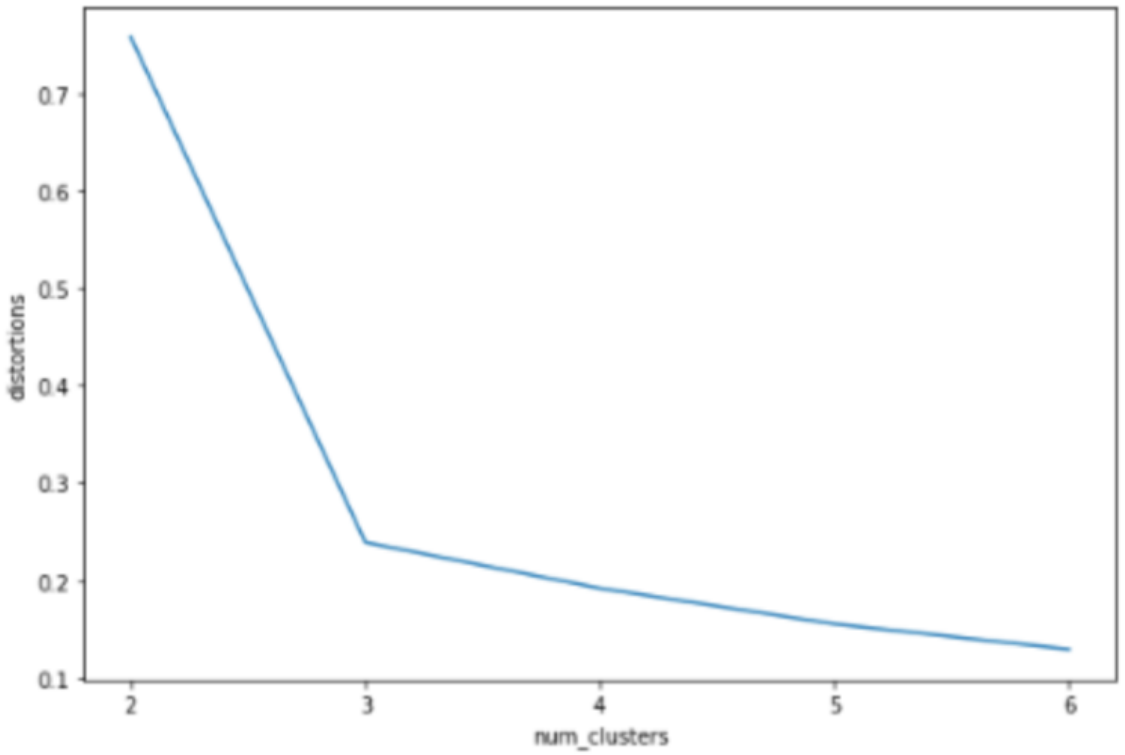
- Elbow plot: plot of the number of clusters and distortion
- Elbow plot helps indicate number of clusters present in data

```
# Declaring variables for use
distortions = []

num_clusters = range(2, 7)
```

```
# Populating distortions for various clusters
for i in num_clusters:
    centroids, distortion = kmeans(df[['scaled_x', 'scaled_y']], i)
    distortions.append(distortion)
```

```
# Plotting elbow plot data
elbow_plot_data = pd.DataFrame({'num_clusters': num_clusters,
                                'distortions': distortions})

sns.lineplot(x='num_clusters', y='distortions',
             data = elbow_plot_data)
plt.show()
```

Elbow method in Python



- Only gives an indication of optimal _k_ (numbers of clusters)
- Does not always pinpoint how many _k_ (numbers of clusters)
- Other methods: average silhouette and gap statistic

→ If its “sharp turn” then it’s elbow → num_clusters = num @ turn
→ If its not sharp turn → then we cannot determine the num_clusters

Limitations of k-means clustering

- How to find the right _K_ (number of clusters)?
- Impact of seeds
- Biased towards equal sized clusters

Impact of seeds

Initialize a random seed

```
from numpy import random
random.seed(12)
```

Seed: `np.array(1000, 2000)`

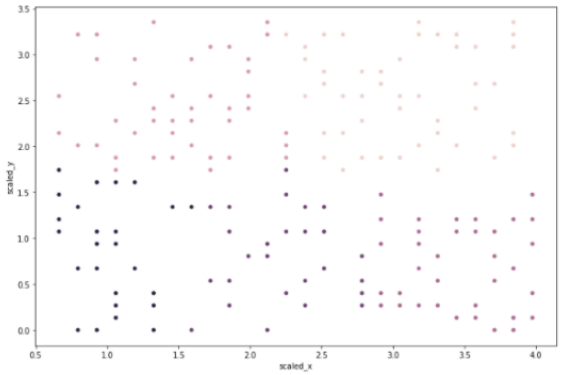
Cluster sizes: 29, 29, 43, 47, 52

Seed: `np.array(1,2,3)`

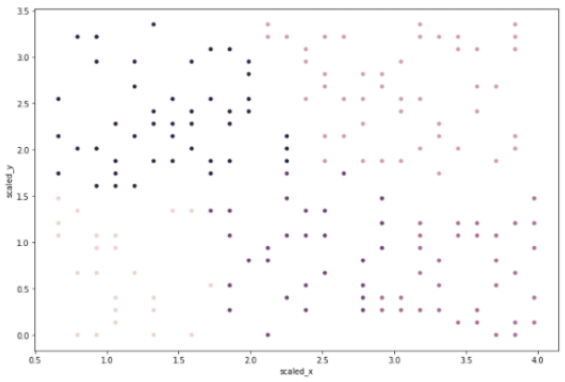
Cluster sizes: 26, 31, 40, 50, 53

Impact of seeds: plots

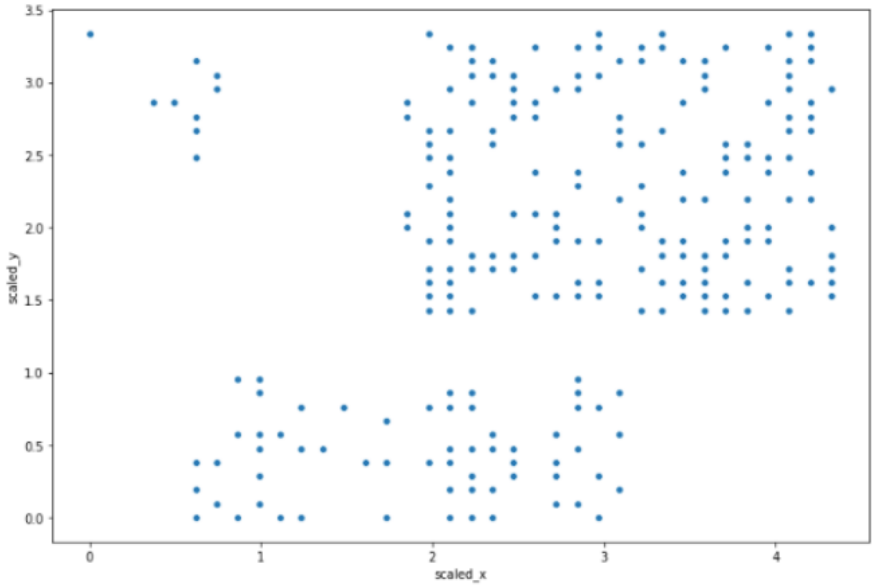
Seed: `np.array(1000, 2000)`



Seed: `np.array(1,2,3)`

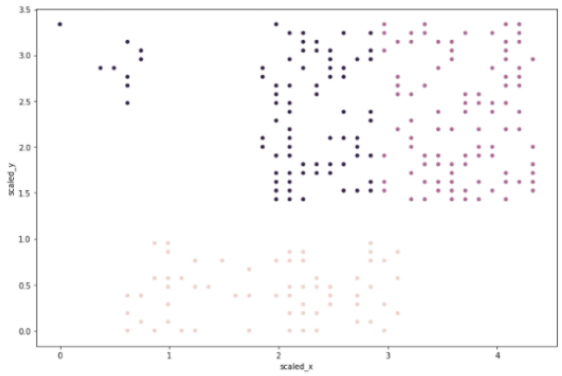


Uniform clusters in k means

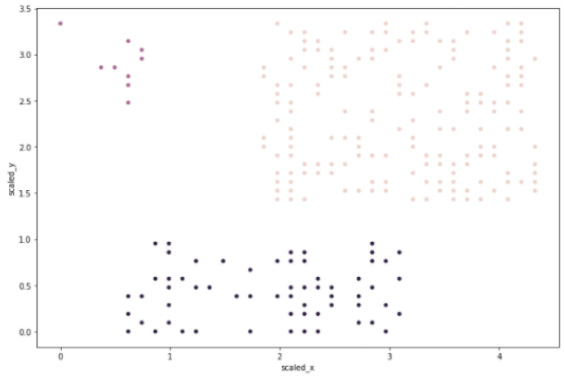


Uniform clusters in k-means: a comparison

K-means clustering with 3 clusters



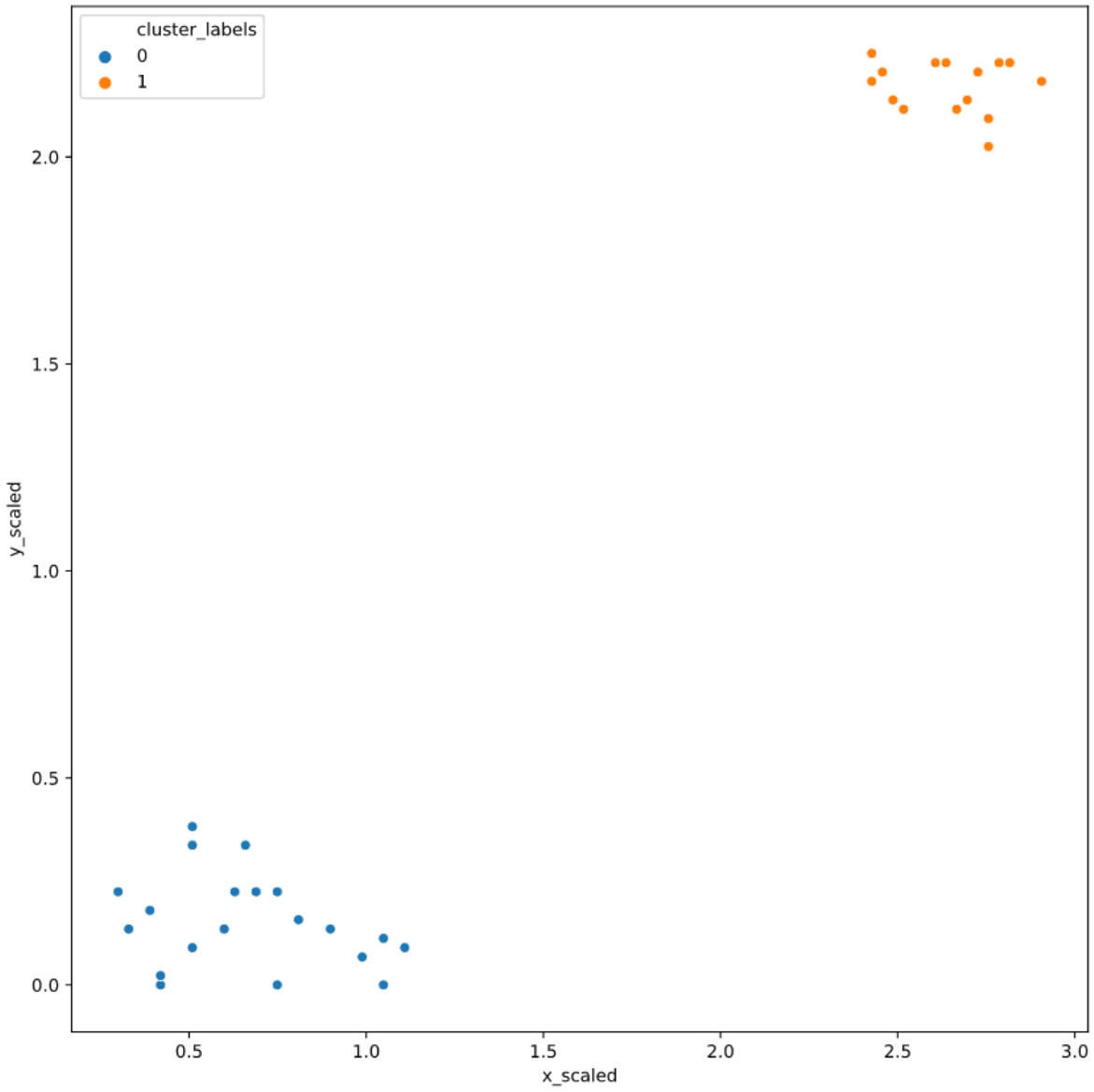
Hierarchical clustering with 3 clusters



- Each technique has its pros and cons
- Consider your data size and patterns before deciding on algorithm
- Clustering is exploratory phase of analysis

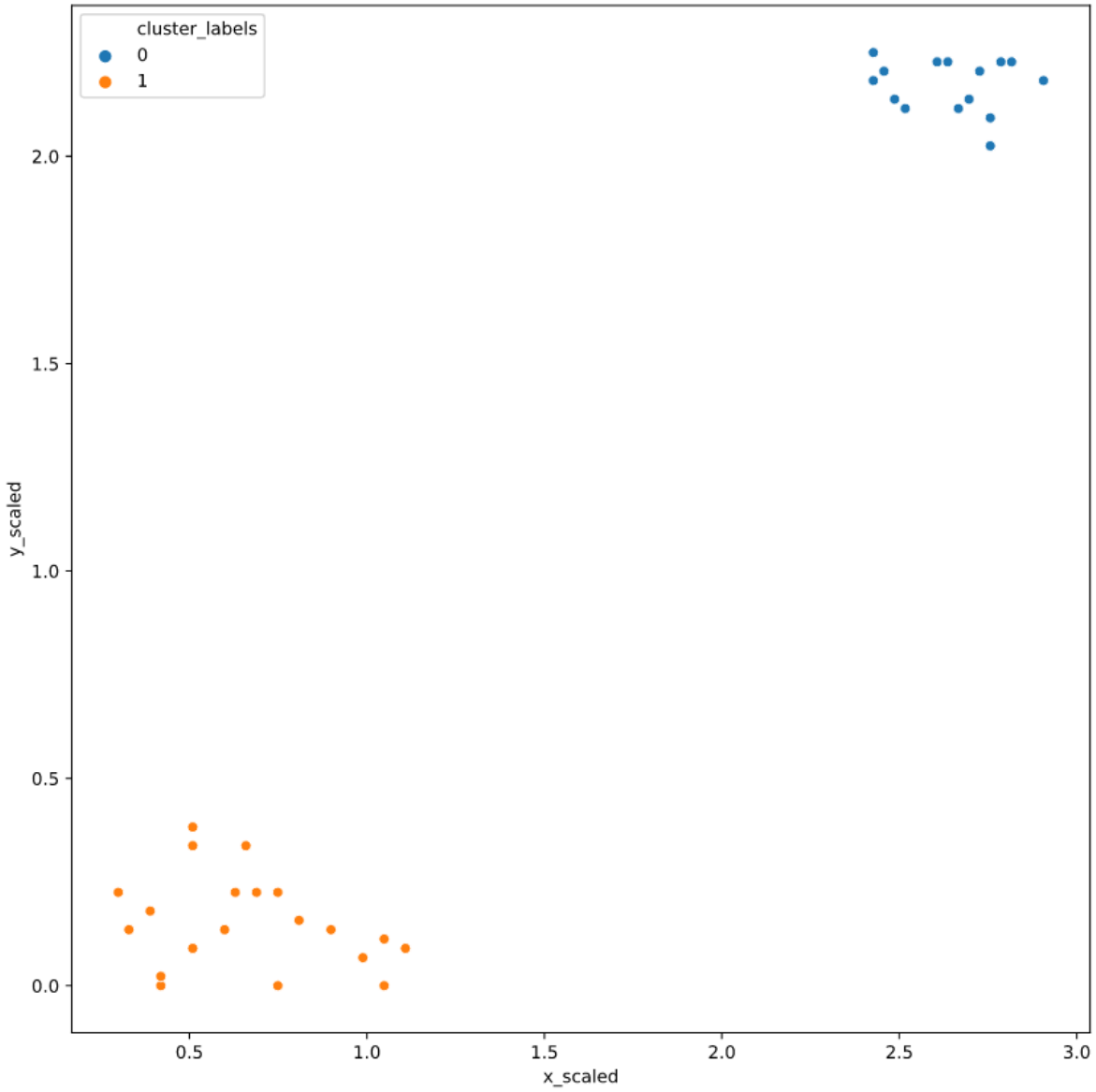
```
# Import random class
from numpy import random
# Initialize seed
random.seed(0)
# Run kmeans clustering
cluster_centers, distortion = kmeans(comic_con[
['x_scaled', 'y_scaled']], 2)
comic_con['cluster_labels'], distortion_list = vq
(comic_con[['x_scaled', 'y_scaled']],
cluster_centers)
# Plot the scatterplot
sns.scatterplot(x='x_scaled', y='y_scaled',
                hue='cluster_labels', data =
comic_con)
plt.show()
```

Impact of seeds on distinct clusters (random.seed(0))



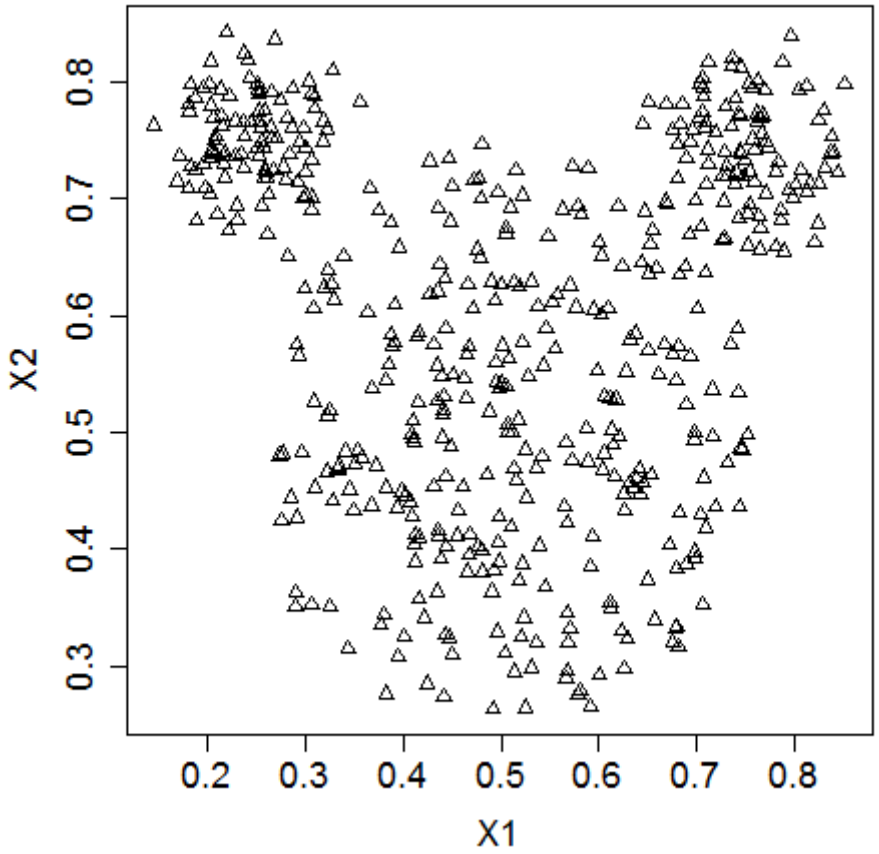
```
# Import random class
from numpy import random
# Initialize seed
random.seed([1,2,1000])
# Run kmeans clustering
cluster_centers, distortion = kmeans(comic_con[
['x_scaled', 'y_scaled']], 2)
comic_con['cluster_labels'], distortion_list = vq
(comic_con[['x_scaled', 'y_scaled']],
cluster_centers)
# Plot the scatterplot
sns.scatterplot(x='x_scaled', y='y_scaled',
                hue='cluster_labels', data =
comic_con)
plt.show()
```

Change random seed to list [1,2,1000]



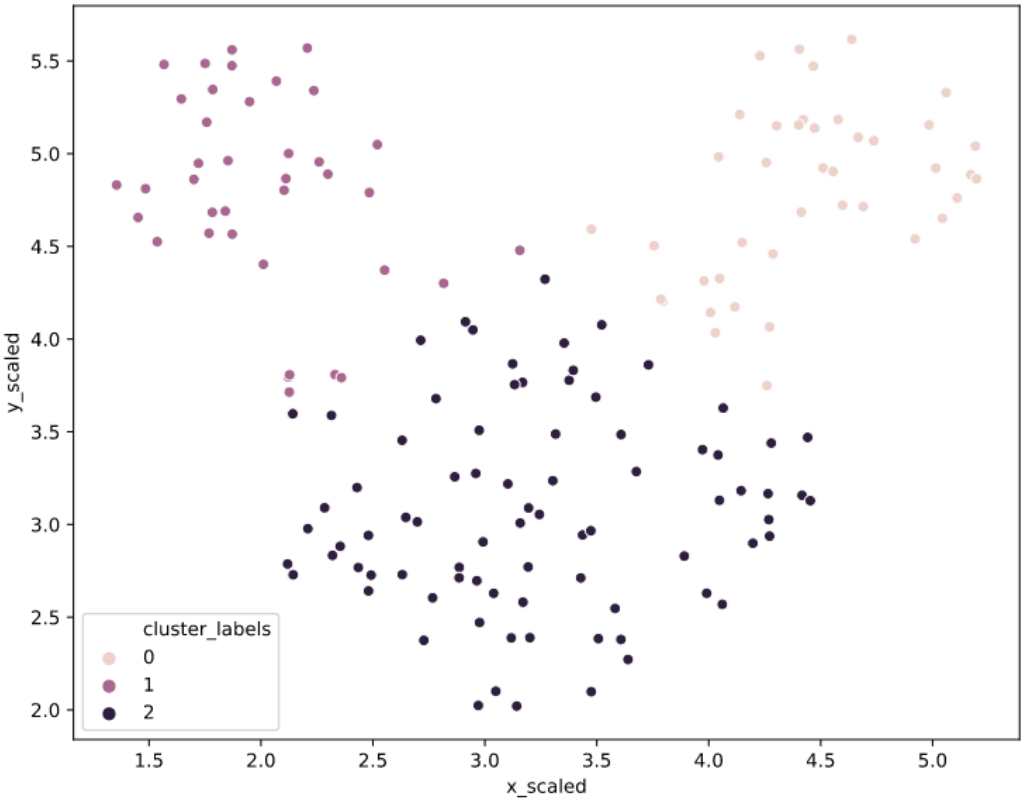
Cluster label swap

Uniform clustering patterns (mouse pattern)
https://www.researchgate.net/figure/Clustering-results-for-the-Mouse-data-set-where-the-black-boxes-represent-the-centroids_fig3_256378655



Clustering results for the Mouse data set, where the black boxes represent the centroids

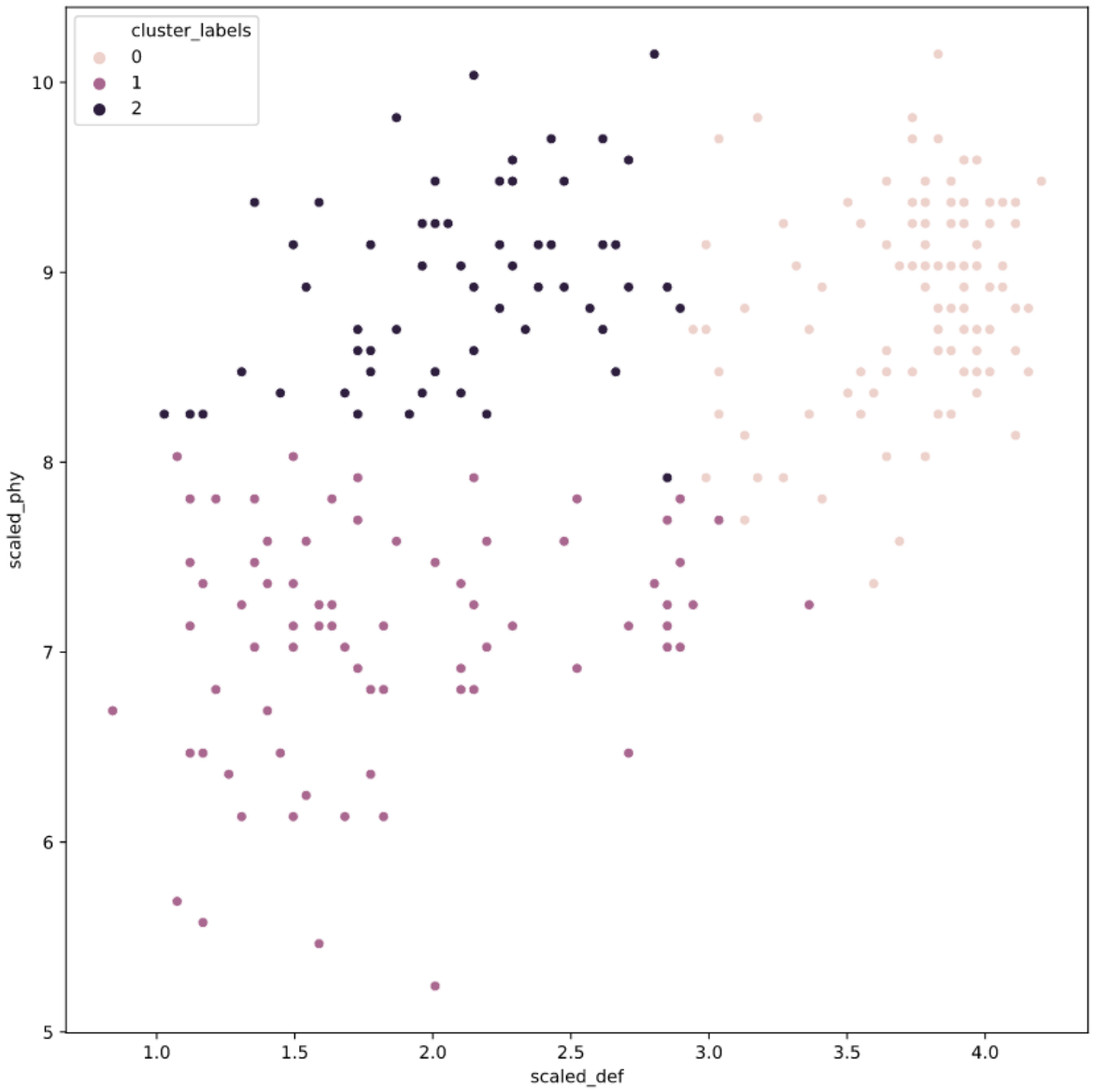
```
# Import the kmeans and vq functions
from scipy.cluster.vq import kmeans, vq
# Generate cluster centers
cluster_centers, distortion = kmeans(mouse[
['x_scaled','y_scaled']], 3)
# Assign cluster labels
mouse['cluster_labels'], distortion_list = vq(mouse[
['x_scaled','y_scaled']], cluster_centers)
# Plot clusters
sns.scatterplot(x='x_scaled', y='y_scaled',
                hue='cluster_labels', data = mouse)
plt.show()
```



Notice that kmeans is unable to capture the three visible clusters clearly, and the two clusters towards the top have taken in some points along the boundary. This happens due to the underlying assumption in kmeans algorithm to minimize distortions which leads to clusters that are similar in terms of area.

```
# Set up a random seed in numpy
random.seed([1000,2000])
# Fit the data into a k-means algorithm
cluster_centers,_ = kmeans(fifa[['scaled_def',
'scaled_phy']], 3)
# Assign cluster labels
fifa['cluster_labels'], _ = vq(fifa[
['scaled_def', 'scaled_phy']], cluster_centers)
# Display cluster centers
print(fifa[['scaled_def', 'scaled_phy',
'cluster_labels']].groupby('cluster_labels').mean
())
# Create a scatter plot through seaborn
sns.scatterplot(x='scaled_def', y='scaled_phy',
hue='cluster_labels', data=fifa)
plt.show()
```

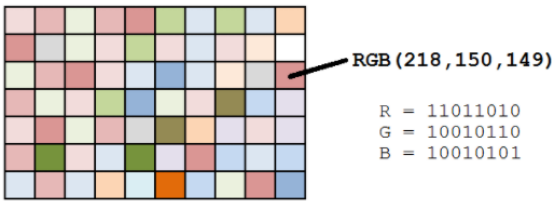
Fifa Defenders



Dominant colors in images

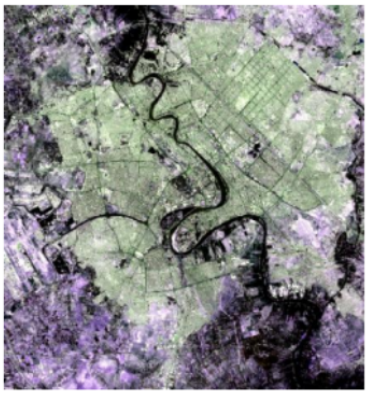
Dominant colors in images

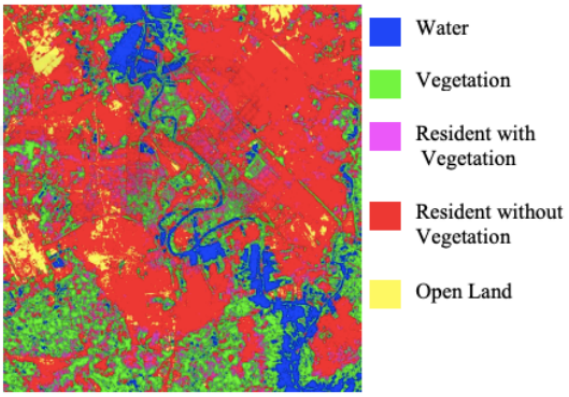
- All images consist of pixels
- Each pixel has three values: *Red*, *Green* and *Blue*
- Pixel color: combination of these RGB values
- Perform k-means on standardized RGB values to find cluster centers
- Uses: Identifying features in satellite images



Source

Feature identification in satellite images





Tools to find dominant colors

- Convert image to pixels: `matplotlib.image.imread`
- Display colors of cluster centers: `matplotlib.pyplot.imshow`

Convert image to RGB matrix

```
import matplotlib.image as img
image = img.imread('sea.jpg')
image.shape
```

```
(475, 764, 3)
```

```
r = []
g = []
b = []

for row in image:
    for pixel in row:
        # A pixel contains RGB values
        temp_r, temp_g, temp_b = pixel
        r.append(temp_r)
        g.append(temp_g)
        b.append(temp_b)
```

Data frame with RGB values

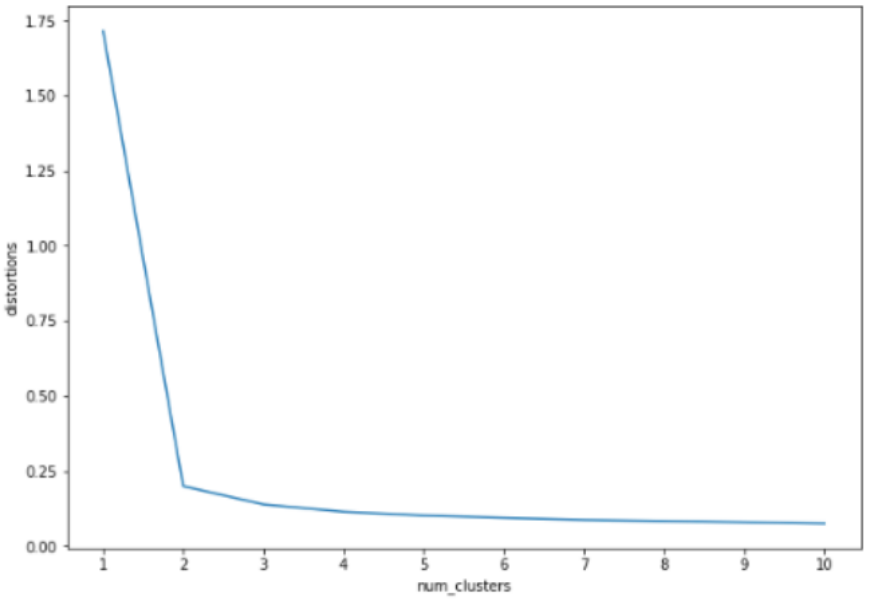
```
pixels = pd.DataFrame({'red': r,
                       'blue': b,
                       'green': g})

pixels.head()
```

red	blue	green
252	255	252
75	103	81
...	...	...

Create an elbow plot

```
distortions = []
num_clusters = range(1, 11)

# Create a list of distortions from the kmeans method
for i in num_clusters:
    cluster_centers, _ = kmeans(pixels[['scaled_red', 'scaled_blue',
                                        'scaled_green']], i)
    distortions.append(distortion)

# Create a data frame with two lists - number of clusters and distortions
elbow_plot = pd.DataFrame({'num_clusters': num_clusters,
                           'distortions': distortions})

# Creat a line plot of num_clusters and distortions
sns.lineplot(x='num_clusters', y='distortions', data = elbow_plot)
plt.xticks(num_clusters)
plt.show()
```

Elbow plot



Find dominant colors

```
cluster_centers, _ = kmeans(pixels[['scaled_red', 'scaled_blue',
                                    'scaled_green']], 2)
```

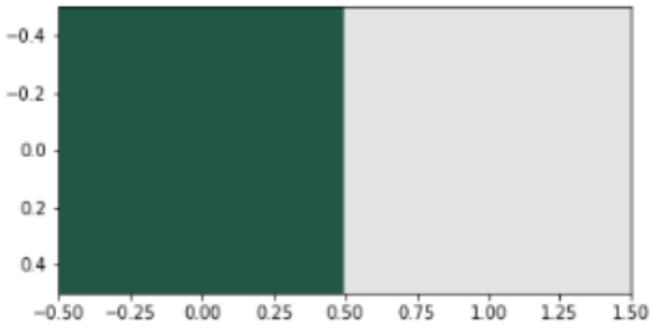
```
colors = []

# Find Standard Deviations
r_std, g_std, b_std = pixels[['red', 'blue', 'green']].std()

# Scale actual RGB values in range of 0-1
for cluster_center in cluster_centers:
    scaled_r, scaled_g, scaled_b = cluster_center
    colors.append((
        scaled_r * r_std/255,
        scaled_g * g_std/255,
        scaled_b * b_std/255
    ))
```

Display dominant colors

```
#Dimensions: 2 x 3 (N X 3 matrix)
print(colors)
```

```
[(0.08192923122023911, 0.34205845943857993, 0.2824002984155429),
 (0.893281510956742, 0.899818770315129, 0.8979114272960784)]
```

```
#Dimensions: 1 x 2 x 3 (1 X N x 3 matrix)
plt.imshow([colors])
plt.show()
```



```
# Import image class of matplotlib
import matplotlib.image as img
# Read batman image and print dimensions
batman_image = img.imread('batman.jpg')
print(batman_image.shape)
# Store RGB values of all pixels in lists r, g and b
for row in batman_image:
    for pixel in row:
        temp_r, temp_g, temp_b = pixel
        r.append(temp_r)
        g.append(temp_g)
        b.append(temp_b)
```

Extract RGB values from image

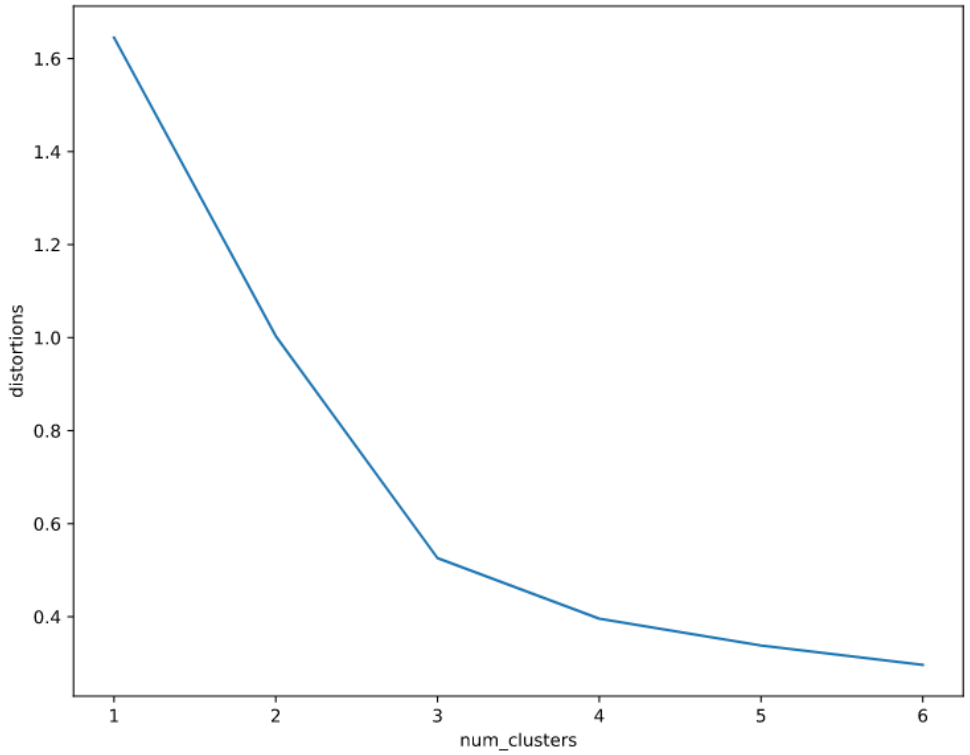
<script.py> output:

(57, 90, 3)

The RGB values are stored in a data frame, batman_df. The RGB values have been standardized used the whiten() function, stored in columns, scaled_red, scaled_blue and scaled_green.

```
distortions = []
num_clusters = range(1, 7)
# Create a list of distortions from the kmeans
function
for i in num_clusters:
    cluster_centers, distortion = kmeans
(batman_df[['scaled_red','scaled_blue',
'scaled_green']],i)
    distortions.append(distortion)
# Create a data frame with two lists,
num_clusters and distortions
elbow_plot = pd.DataFrame(
{'num_clusters':num_clusters,
'distortions':distortions})
# Create a line plot of num_clusters and
distortions
sns.lineplot(x='num_clusters', y='distortions',
data = elbow_plot)
plt.xticks(num_clusters)
plt.show()
```

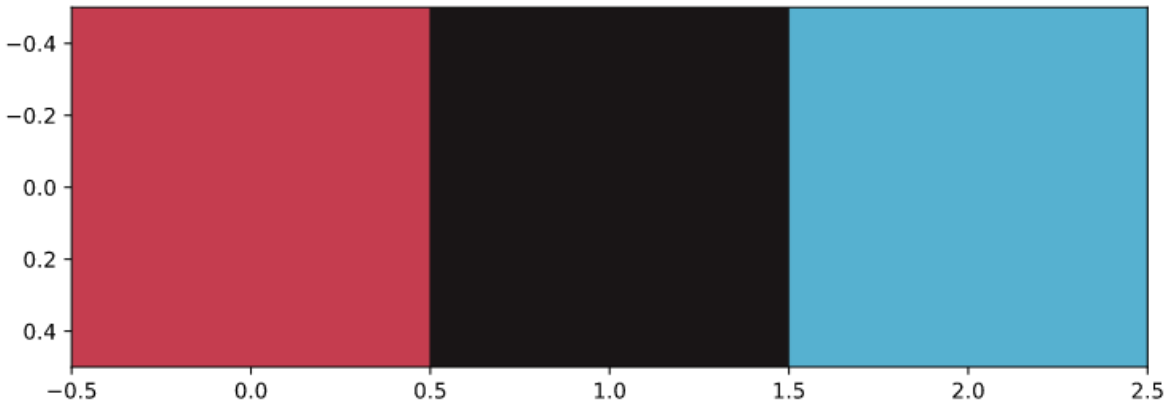
How many dominant color (find using Elbow plot)



3 clusters → 3 colors

```
# Get standard deviations of each color
r_std, g_std, b_std = batman_df[['red', 'green',
'blue']].std()
for cluster_center in cluster_centers:
    scaled_r, scaled_g, scaled_b = cluster_center
    # Convert each standardized value to scaled
value
    colors.append((
        scaled_r * r_std / 255,
        scaled_g * g_std / 255,
        scaled_b * b_std / 255
    ))
# Display colors of cluster centers
plt.imshow([colors])
plt.show()
```

Display dominant colors



Document Clustering

1. Clean data before processing
2. Determine the importance of the terms in a document (in TF-IDF matrix)
3. Cluster the TF-IDF matrix
4. Find top terms, documents in each cluster

TF = Term Frequency, IDF = Inverse Document Frequency

https://lukkiddd.com/tf-idf-%E0%B8%84%E0%B8%B3%E0%B9%84%E0%B8%AB%E0%B8%99%E0%B8%AA%E0%B8%B3%E0%B8%84%E0%B8%B1%E0%B8%8D%E0%B8%99%E0%B8%B0-dd1e1568312e

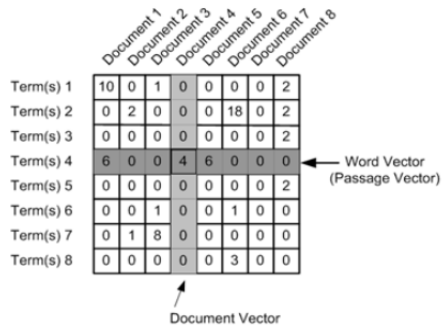
Clean and tokenize data

- Convert text into smaller parts called tokens, clean data for processing

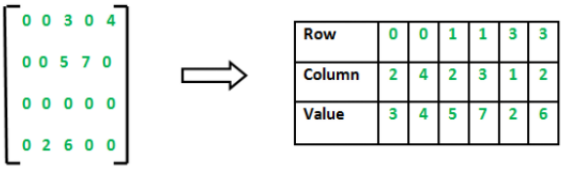
```
from nltk.tokenize import word_tokenize
import re

def remove_noise(text, stop_words = []):
    tokens = word_tokenize(text)
    cleaned_tokens = []
    for token in tokens:
        token = re.sub('[^A-Za-z0-9]+', '', token)
        if len(token) > 1 and token.lower() not in stop_words:
            # Get lowercase
            cleaned_tokens.append(token.lower())
    return cleaned_tokens
```

Document term matrix and sparse matrices

- Document term matrix formed
- Most elements in matrix are zeros



- Sparse matrix is created



Source

TF-IDF (Term Frequency - Inverse Document Frequency)

- A weighted measure: evaluate how important a word is to a document in a collection

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer(max_df=0.8, max_features=50,
                                   min_df=0.2, tokenizer=remove_noise)
tfidf_matrix = tfidf_vectorizer.fit_transform(data)
```

Clustering with sparse matrix

- `kmeans()` in SciPy does not support sparse matrices
- Use `.todense()` to convert to a matrix

```
cluster_centers, distortion = kmeans(tfidf_matrix.todense(), num_clusters)
```

Top terms per cluster

- Cluster centers: lists with a size equal to the number of terms
- Each value in the cluster center is its importance
- Create a dictionary and print top terms

```
terms = tfidf_vectorizer.get_feature_names()

for i in range(num_clusters):
    center_terms = dict(zip(terms, list(cluster_centers[i])))
    sorted_terms = sorted(center_terms, key=center_terms.get, reverse=True)
    print(sorted_terms[:3])
```

```
['room', 'hotel', 'staff']

['bad', 'location', 'breakfast']
```

- Work with hyperlinks, emoticons etc.
- Normalize words (run, ran, running -> run)
- `.todense()` may not work with large datasets

```
# Import TfidfVectorizer class from sklearn
from sklearn.feature_extraction.text import TfidfVectorizer
# Initialize TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer(max_df=0.75, max_features=50, min_df=0.1, tokenizer=remove_noise)
# Use the .fit_transform() method on the list plots
tfidf_matrix = tfidf_vectorizer.fit_transform(plots)
```

TF-IDF of movie plots

```
num_clusters = 2
# Generate cluster centers through the kmeans function
cluster_centers, distortion = kmeans(tfidf_matrix.todense(), num_clusters)
# Generate terms from the tfidf_vectorizer object
terms = tfidf_vectorizer.get_feature_names()
for i in range(num_clusters):
    # Sort the terms and print top 3 terms
    center_terms = dict(zip(terms, cluster_centers[i]))
    sorted_terms = sorted(center_terms, key=center_terms.get, reverse=True)
    print(sorted_terms[:3])
```

Top terms in movie clusters

<script.py> output:
['father', 'back', 'one']
['police', 'man', 'killed']

Clustering with multiple features

Basic checks

```
# Cluster centers
print(fifa.groupby('cluster_labels')[['scaled_heading_accuracy',
    'scaled_volleys', 'scaled_finishing']].mean())
```

cluster_labels	scaled_heading_accuracy	scaled_volleys	scaled_finishing
0	3.21	2.83	2.76
1	0.71	0.64	0.58

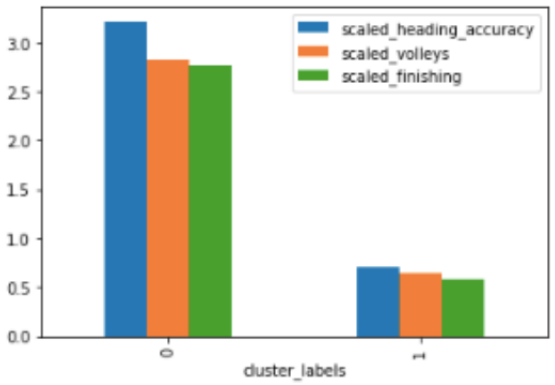
```
# Cluster sizes
print(fifa.groupby('cluster_labels')['ID'].count())
```

cluster_labels	count
0	886
1	114

Visualizations

- Visualize cluster centers
- Visualize other variables for each cluster

```
# Plot cluster centers
fifa.groupby('cluster_labels') \
  [scaled_features].mean()
  .plot(kind='bar')
plt.show()
```



Top items in clusters

```
# Get the name column of top 5 players in each cluster
for cluster in fifa['cluster_labels'].unique():
    print(cluster, fifa[fifa['cluster_labels'] == cluster]['name'].values[:5])
```

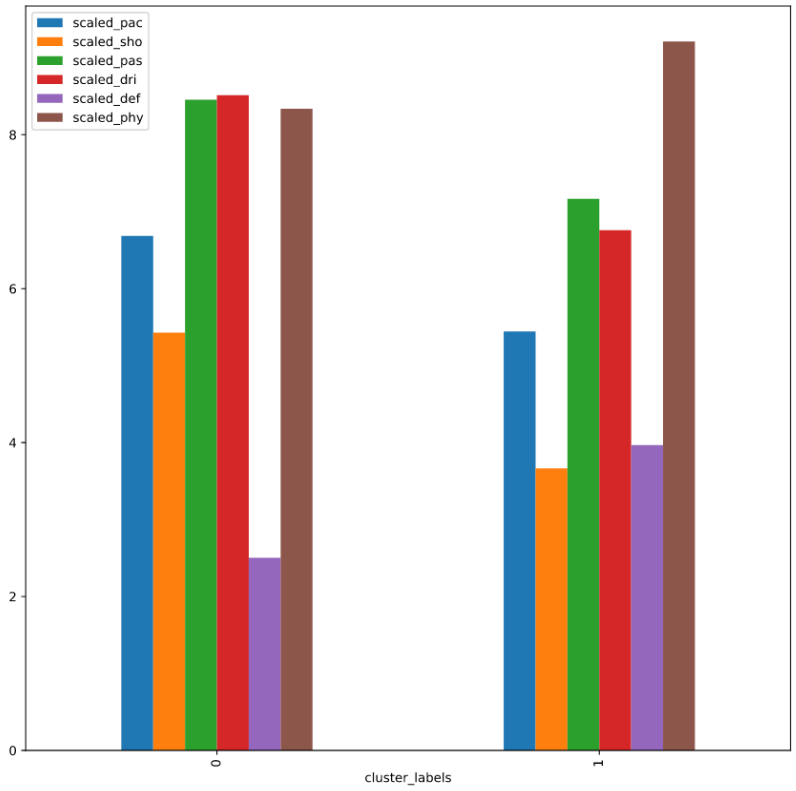
Cluster Label	Top Players
0	['Cristiano Ronaldo' 'L. Messi' 'Neymar' 'L. Suárez' 'R. Lewandowski']
1	['M. Neuer' 'De Gea' 'G. Buffon' 'T. Courtois' 'H. Lloris']

Feature reduction

- Factor analysis
- Multidimensional scaling

```
# Create centroids with kmeans for 2 clusters
cluster_centers,_ = kmeans(fifa[scaled_features], 2)
# Assign cluster labels and print cluster centers
fifa['cluster_labels'], _ = vq(fifa[scaled_features],
cluster_centers)
print(fifa.groupby('cluster_labels')[scaled_features].mean())
# Plot cluster centers to visualize clusters
fifa.groupby('cluster_labels')[scaled_features].mean().plot
(legend=True, kind='bar')
plt.show()
# Get the name column of first 5 players in each cluster
for cluster in fifa['cluster_labels'].unique():
    print(cluster, fifa[fifa['cluster_labels'] == cluster]
['name'].values[:5])
```

FIFA 18: what makes a complete player?



<script.py> output:

```
                scaled_pac  scaled_sho  scaled_pas  scaled_dri  scaled_def  \
cluster_labels
0                     6.68        5.43        8.46        8.51        2.50
1                     5.44        3.66        7.17        6.76        3.97

                scaled_phy
cluster_labels
0                     8.34
1                     9.21
0 ['Cristiano Ronaldo' 'L. Messi' 'Neymar' 'L. Suárez' 'M. Neuer']
1 ['Sergio Ramos' 'G. Chiellini' 'D. Godín' 'Thiago Silva' 'M. Hummels']
```

Dimensionality Reduction in Python

Why reduce dimensionality?

Dataset will be

- less complex
- require less disk space
- require less computation time
- have lower chance of model overfitting

income	age	favorite color
10000	18	Black
50000	47	Blue
20000	40	Blue
30000	29	Green
20000	22	Purple

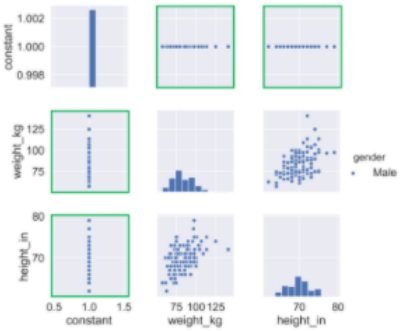
→

income	age
10000	18
50000	47
20000	40
30000	29
20000	22

```
insurance_df.drop('favorite color', axis=1)
```

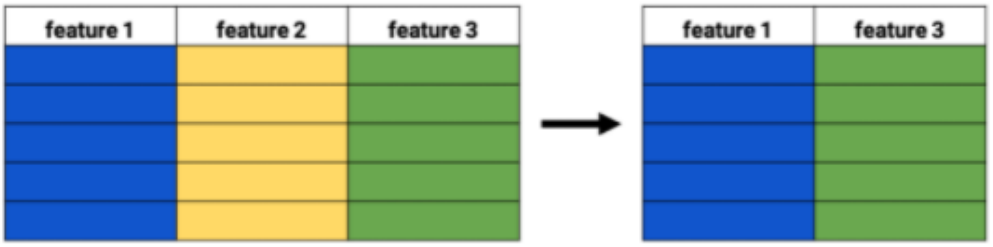
Feature selection

```
sns.pairplot(ansur_df, hue="gender", diag_kind='hist')
```

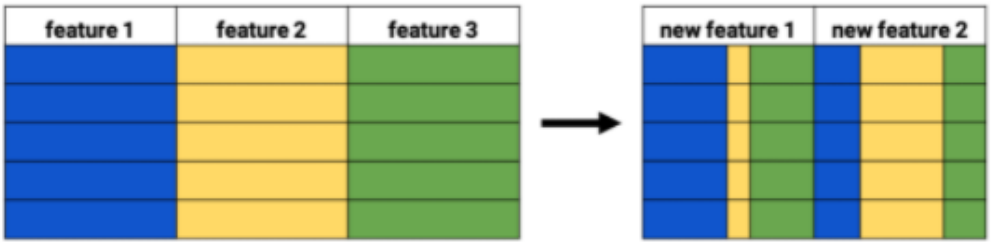


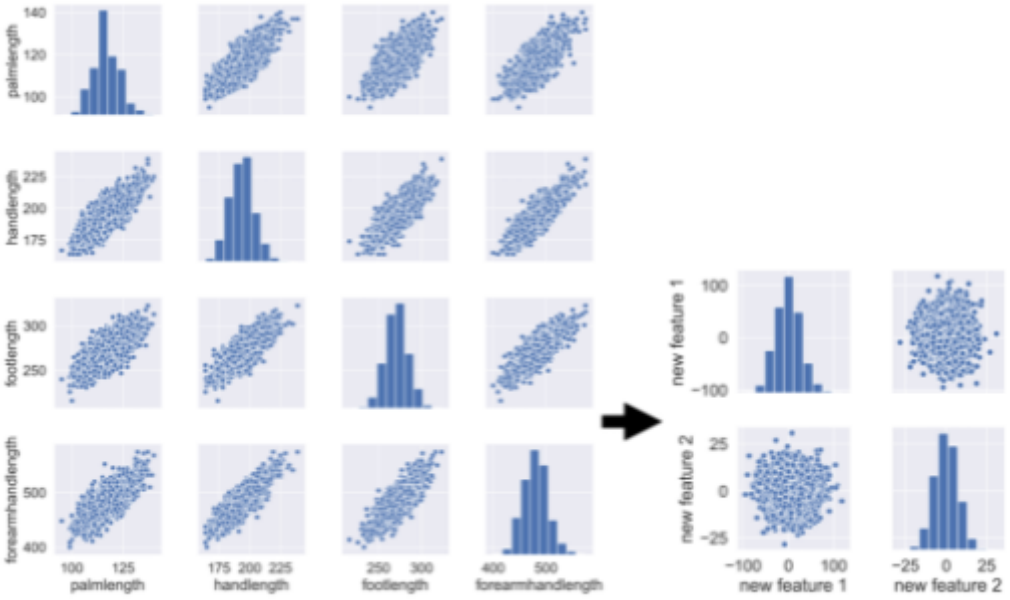
Pairplot in ANSUR data

Feature selection



Feature extraction





Feature extraction - Example

t-SNE visualization of high-dimensional data

```
df.shape
```

```
(1986, 99)
```

```
non_numeric = ['BMI_class', 'Height_class',
               'Gender',  'Component', 'Branch']

df_numeric = df.drop(non_numeric, axis=1)

df_numeric.shape
```

```
(1986, 94)
```

t-SNE on female ANSUR dataset

```
from sklearn.manifold import TSNE

m = TSNE(learning_rate=50)
```

```
tsne_features = m.fit_transform(df_numeric)

tsne_features[1:4,:]
```

```
array([[-37.962185,  15.066088],
       [-21.873512,  26.334448],
       [ 13.97476 ,  22.590828]], dtype=float32)
```

Fitting t-SNE

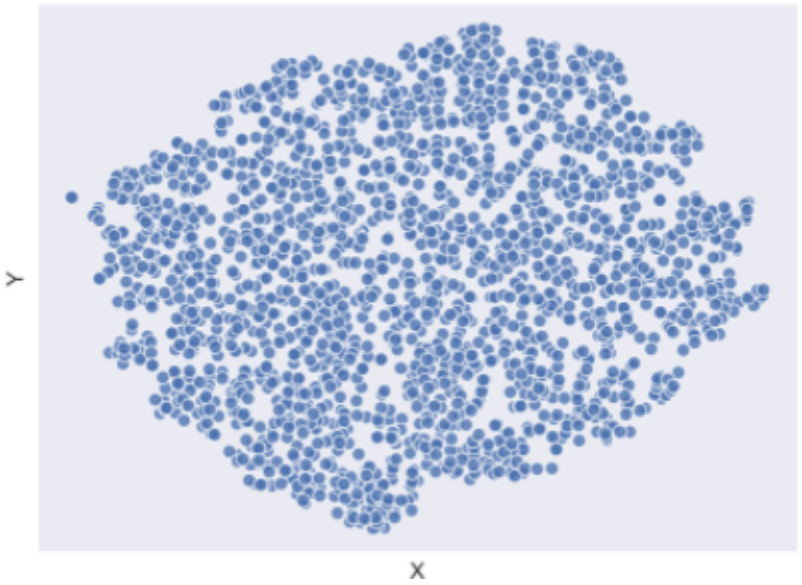
```
tsne_features[1:4,:]
```

```
array([[-37.962185,  15.066088],
       [-21.873512,  26.334448],
       [ 13.97476 ,  22.590828]], dtype=float32)
```

```
df['x'] = tsne_features[:,0]

df['y'] = tsne_features[:,1]
```

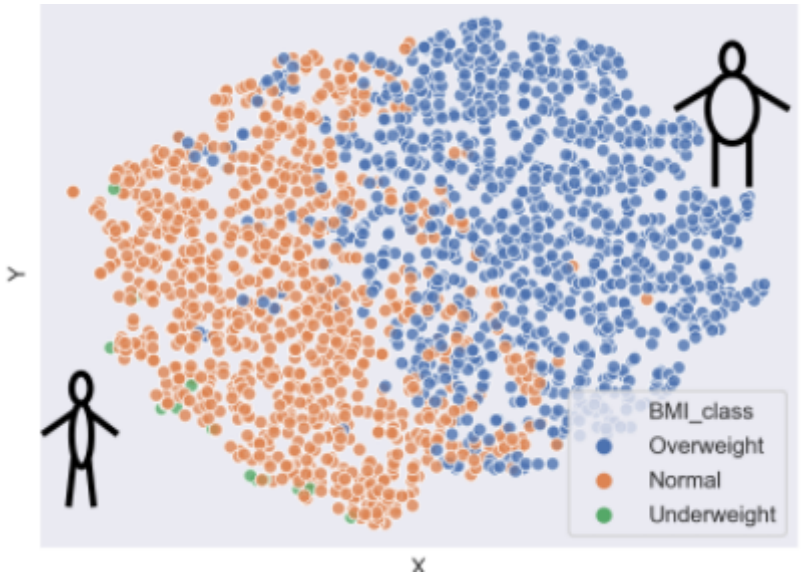
Assigning t-SNE features to our dataset

```
import seaborn as sns

sns.scatterplot(x="x", y="y", data=df)

plt.show()
```



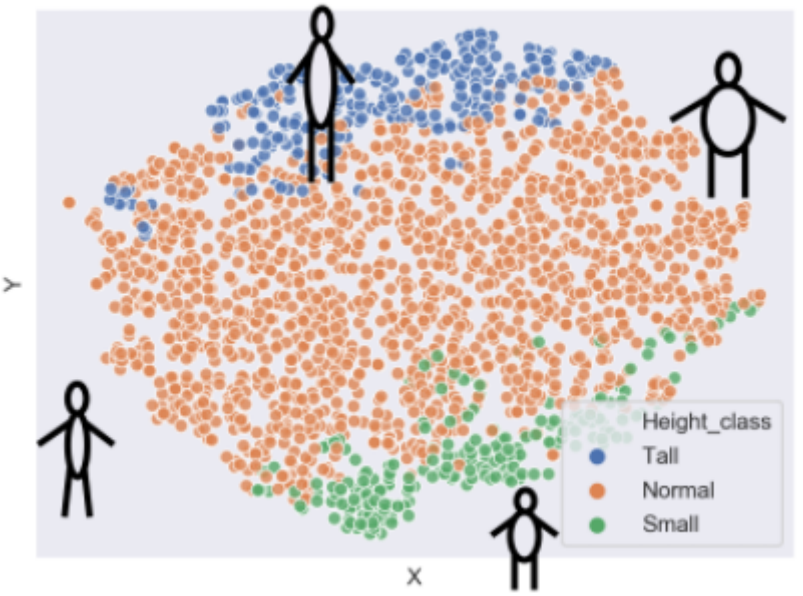
Plotting t-SNE

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.scatterplot(x="x", y="y", hue='BMI_class', data=df)

plt.show()
```



Coloring points according to BMI category

```
import seaborn as sns

import matplotlib.pyplot as plt

sns.scatterplot(x="x", y="y", hue='Height_class', data=df)

plt.show()
```



Coloring points according to height category

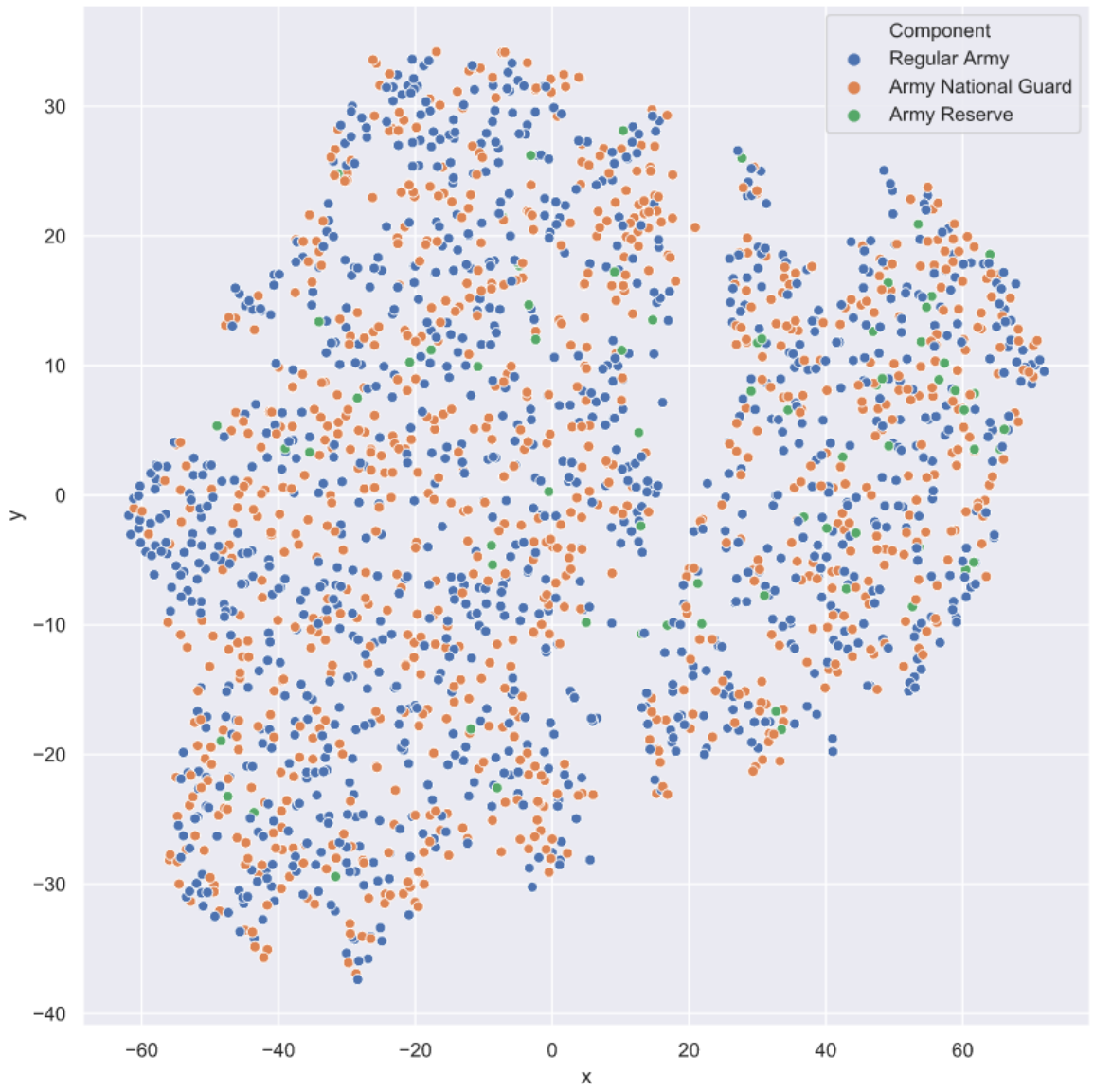
What is a good use case to use t-SNE?
When you want to visually explore the patterns in a high dimensional dataset.

```
# Non-numerical columns in the dataset
non_numeric = ['Branch', 'Gender', 'Component']
# Drop the non-numerical columns from df
df_numeric = df.drop(non_numeric, axis=1)
# Create a t-SNE model with learning rate 50
m = TSNE(learning_rate=50)
# Fit and transform the t-SNE model on the numeric dataset
tsne_features = m.fit_transform(df_numeric)
print(tsne_features.shape)
```

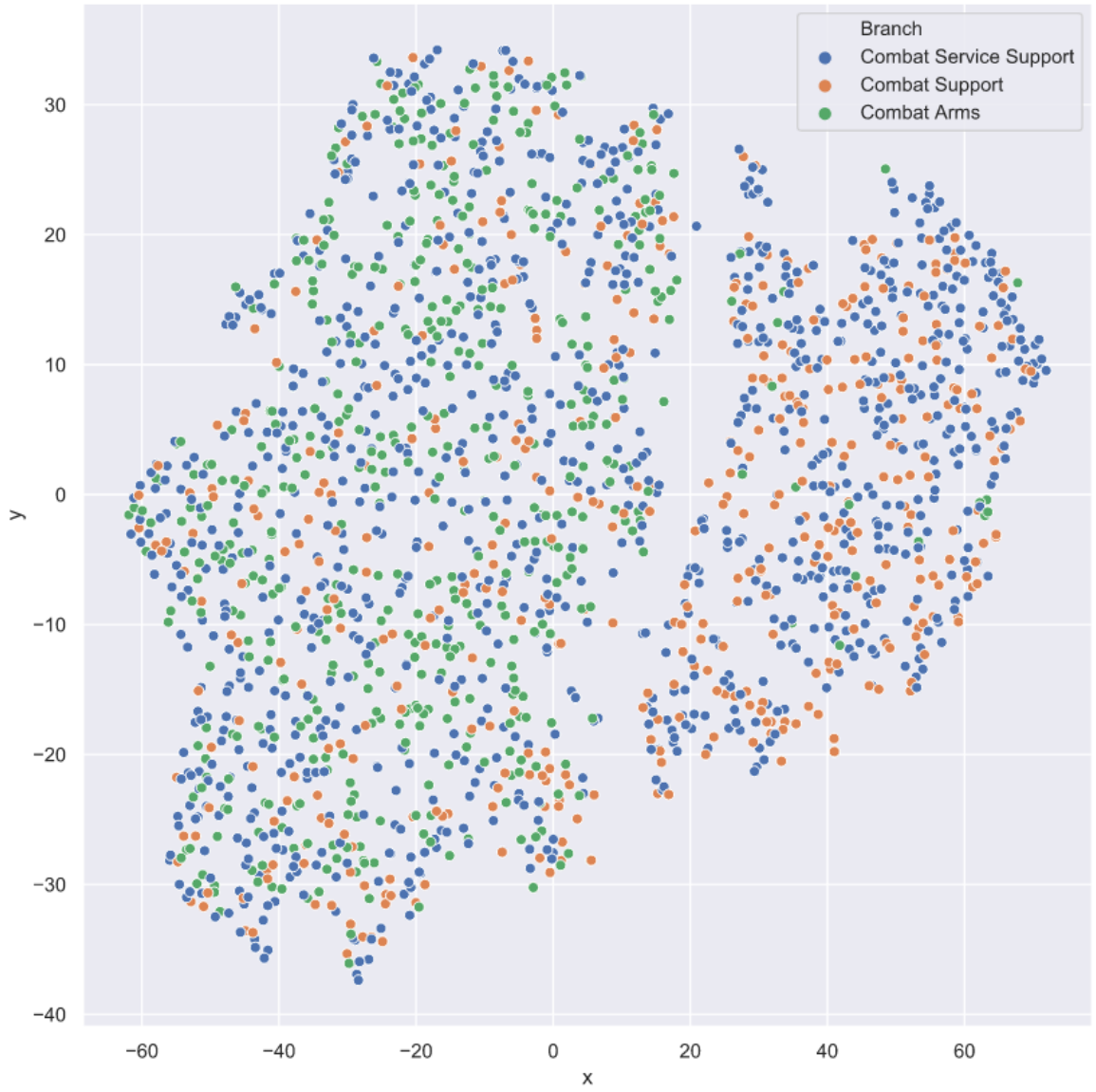
Fitting t-SNE to the ANSUR data.

```
# Color the points by Gender
sns.scatterplot(x="x", y="y", hue='Gender',
data=df)
# Show the plot
plt.show()
```

t-SNE visualisation of dimensionality



hue = ‘Component’

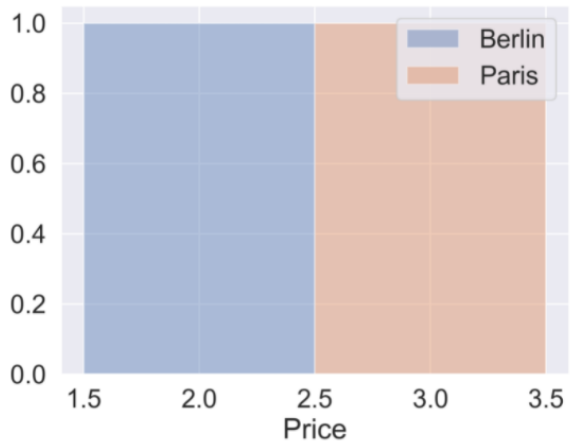

Branch
Combat Service Support
Combat Support
Combat Arms
30
20
10
0
−10
−20
−30
−40
y
−60
−40
−20
0
20
40
60
x

hue = ‘Branch’

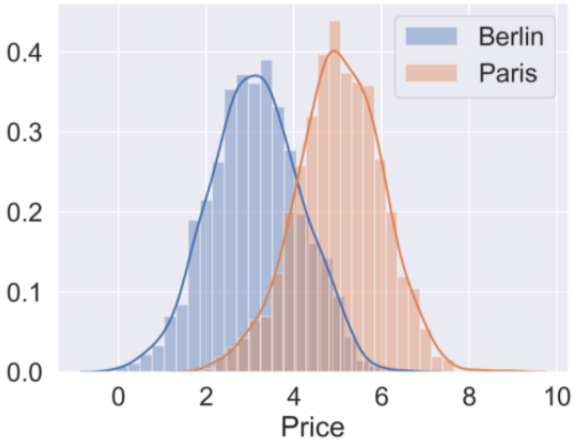
hue = ‘Gender’

The curse of dimensionality

City	Price
Berlin	2
Paris	3



City	Price
Berlin	2.0
Berlin	3.1
Berlin	4.3
Paris	3.0
Paris	5.2
...	...



Separate the feature we want to predict from the ones to train the model on.

```
y = house_df['City']

X = house_df.drop('City', axis=1)
```

Perform a 70% train and 30% test data split

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

Create a Support Vector Machine Classifier and fit to training data

```
from sklearn.svm import SVC

svc = SVC()

svc.fit(X_train, y_train)
```

```
from sklearn.metrics import accuracy_score

print(accuracy_score(y_test, svc.predict(X_test)))
```

```
0.826
```

```
print(accuracy_score(y_train, svc.predict(X_train)))
```

```
0.832
```

Adding features

City	Price	n_floors	n_bathroom	surface_m2
Berlin	2.0	1	1	190
Berlin	3.1	2	1	187
Berlin	4.3	2	2	240
Paris	3.0	2	1	170
Paris	5.2	2	2	290
...	...	...	...	...

```
# Import train_test_split()
from sklearn.model_selection import train_test_split
# Select the Gender column as the feature to be predicted (y)
y = ansur_df['Gender']
# Remove the Gender column to create the training data
X = ansur_df.drop('Gender', axis=1)
# Perform a 70% train and 30% test data split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
print("{} rows in test set vs. {} in training set. {} Features.".format(X_test.shape[0], X_train.shape[0],
X_test.shape[1]))
```

Train - test split

```
# Import SVC from sklearn.svm and accuracy_score from sklearn.metrics
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
# Create an instance of the Support Vector Classification class
svc = SVC()
# Fit the model to the training data
svc.fit(X_train, y_train)
# Calculate accuracy scores on both train and test data
accuracy_train = accuracy_score(y_train, svc.predict(X_train))
accuracy_test = accuracy_score(y_test, svc.predict(X_test))
print("{0:.1%} accuracy on test set vs. {1:.1%} on training set".format(accuracy_test, accuracy_train))
```

Fitting and testing the model

<script.py> output:

49.7% accuracy on test set vs. 100.0% on training set

```
# Assign just the 'neckcircumferencebase' column from ansur_df to X
X = ansur_df[['neckcircumferencebase']]
# Split the data, instantiate a classifier and fit the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
svc = SVC()
svc.fit(X_train, y_train)
# Calculate accuracy scores on both train and test data
accuracy_train = accuracy_score(y_train, svc.predict(X_train))
accuracy_test = accuracy_score(y_test, svc.predict(X_test))
print("{0:.1%} accuracy on test set vs. {1:.1%} on training set".format(accuracy_test, accuracy_train))
```

Accuracy after dimensionality reduction

<script.py> output:

93.3% accuracy on test set vs. 94.9% on training set

On the full dataset the model is rubbish but with a single feature we can make good predictions? This is an example of the curse of dimensionality! The model badly overfits when we feed it too many features. It overlooks that neck circumference by itself is pretty different for males and females.

Features with missing values or little variance

Creating a feature selector

```
print(ansur_df.shape)
```

```
(6068, 94)
```

```
from sklearn.feature_selection import VarianceThreshold

sel = VarianceThreshold(threshold=1)
sel.fit(ansur_df)

mask = sel.get_support()
print(mask)
```

```
array([ True,  True, ..., False,  True])
```

Applying a feature selector

```
print(ansur_df.shape)
```

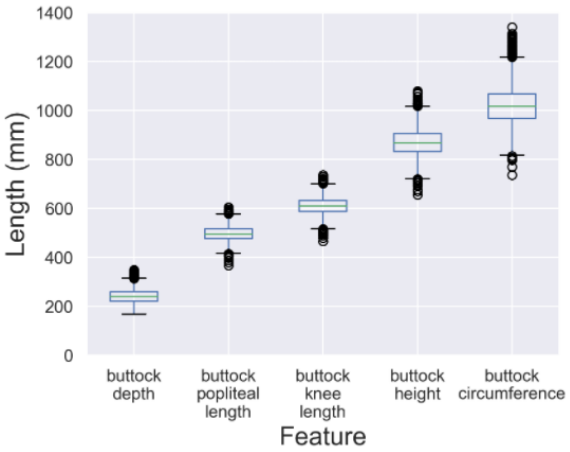
```
(6068, 94)
```

```
reduced_df = ansur_df.loc[:, mask]
print(reduced_df.shape)
```

```
(6068, 93)
```

Variance selector caveats

```
buttock_df.boxplot()
```



Normalizing the variance

```
from sklearn.feature_selection import VarianceThreshold

sel = VarianceThreshold(threshold=0.005)

sel.fit(ansur_df / ansur_df.mean())
mask = sel.get_support()
reduced_df = ansur_df.loc[:, mask]
print(reduced_df.shape)
```

```
(6068, 45)
```

Missing value selector

Name	Type 1	Type 2	Total	HP	Attack	Defense
Bulbasaur	Grass	Poison	318	45	49	49
Ivysaur	Grass	Poison	405	60	62	63
Venusaur	Grass	Poison	525	80	82	83
Charmander	Fire	NaN	309	39	52	43
Charmeleon	Fire	NaN	405	58	64	58

Identifying missing values

```
pokemon_df.isna()
```

Name	Type 1	Type 2	Total	HP	Attack	Defense
False	False	False	False	False	False	False
False	False	False	False	False	False	False
False	False	False	False	False	False	False
False	False	True	False	False	False	False
False	False	True	False	False	False	False

Counting missing values

```
pokemon_df.isna().sum()
```

```
Name          0
Type 1        0
Type 2      386
Total         0
HP            0
Attack        0
Defense       0
dtype: int64
```

```
pokemon_df.isna().sum() / len(pokemon_df)
```

```
Name        0.00
Type 1      0.00
Type 2      0.48
Total       0.00
HP          0.00
Attack      0.00
Defense     0.00
dtype: float64
```

Applying a missing value threshold

```
# Fewer than 30% missing values = True value
mask = pokemon_df.isna().sum() / len(pokemon_df) < 0.3
print(mask)
```

```
Name         True
Type 1       True
Type 2      False
Total        True
HP           True
Attack       True
Defense      True
dtype: bool
```

```
reduced_df = pokemon_df.loc[:, mask]

reduced_df.head()
```

Name	Type 1	Total	HP	Attack	Defense
Bulbasaur	Grass	318	45	49	49
Ivysaur	Grass	405	60	62	63
Venusaur	Grass	525	80	82	83
Charmander	Fire	309	39	52	43
Charmeleon	Fire	405	58	64	58

```
from sklearn.feature_selection import VarianceThreshold
# Create a VarianceThreshold feature selector
sel = VarianceThreshold(threshold=0.001)
# Fit the selector to normalized head_df
sel.fit(head_df / head_df.mean())
# Create a boolean mask
mask = sel.get_support()
# Apply the mask to create a reduced dataframe
reduced_df = head_df.loc[:, mask]
print("Dimensionality reduced from {} to {}.".format(head_df.shape[1], reduced_df.shape[1]))
```

Features with low variance

<script.py> output:
Dimensionality reduced from 6 to 4.

```
school_df.isna().sum() / len(school_df)

x           0.000000

y           0.000000
objectid_1    0.000000
objectid      0.000000
bldg_id       0.000000
bldg_name     0.000000
address       0.000000
city          0.000000
zipcode       0.000000
csp_sch_id    0.000000
sch_id        0.000000
sch_name      0.000000
sch_label     0.000000
sch_type      0.000000
shared        0.877863
complex       0.984733
label         0.000000
tlt           0.000000
pl            0.000000
point_x       0.000000
point_y       0.000000
dtype: float64
```

So, max range roughly 0.9 - 1.0

```
# Create a boolean mask on whether each feature less than 50% missing values.
mask = school_df.isna().sum() / len(school_df) < 0.5
# Create a reduced dataset by applying the mask
reduced_df = school_df.loc[:, mask]
print(school_df.shape)
print(reduced_df.shape)
```

Removing features with many missing values

<script.py> output:

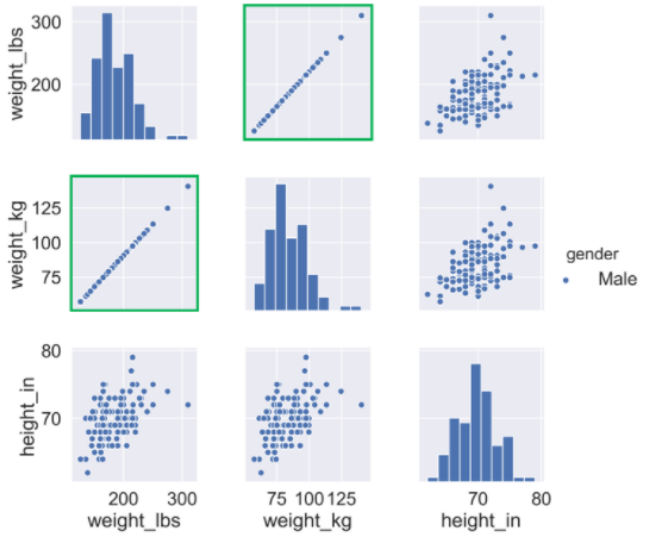
(131, 21)

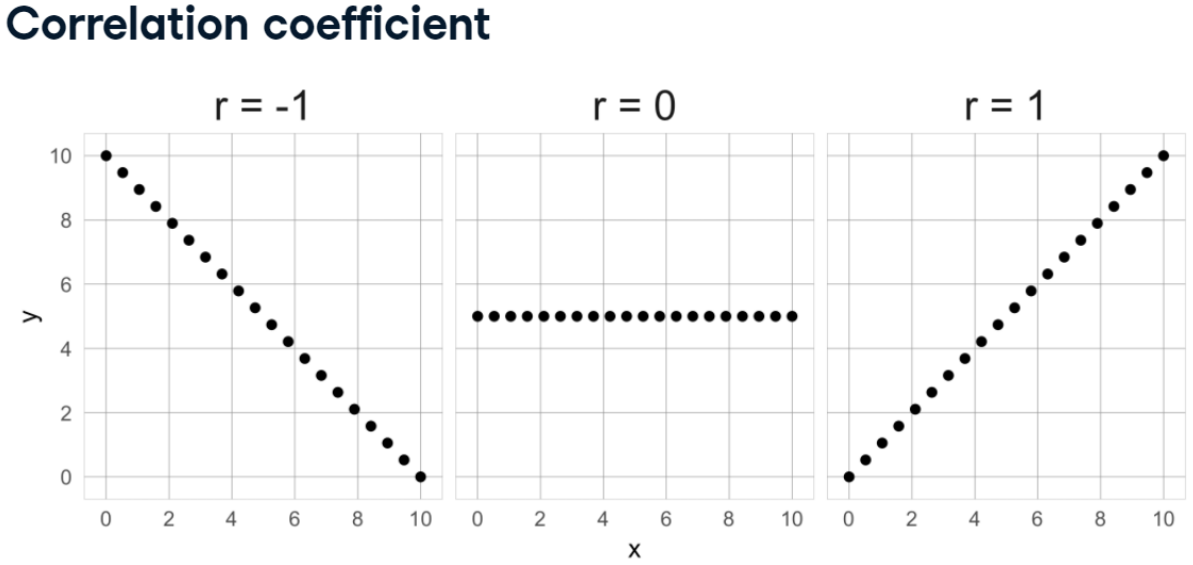
(131, 19)

Great! The number of features went down from 21 to 19.

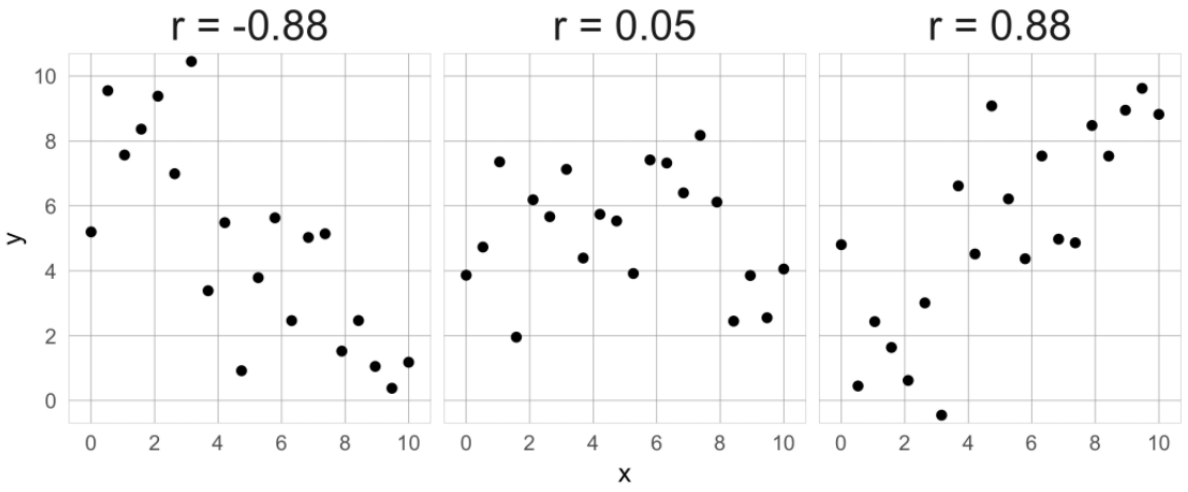
Pairwise correlation

In summary, using pairwise correlation allows us to detect highly correlated features which bring no new information to the dataset. Since these features only add to model complexity, increase the chance of overfitting, and require more computations, they should be dropped.





Correlation coefficient



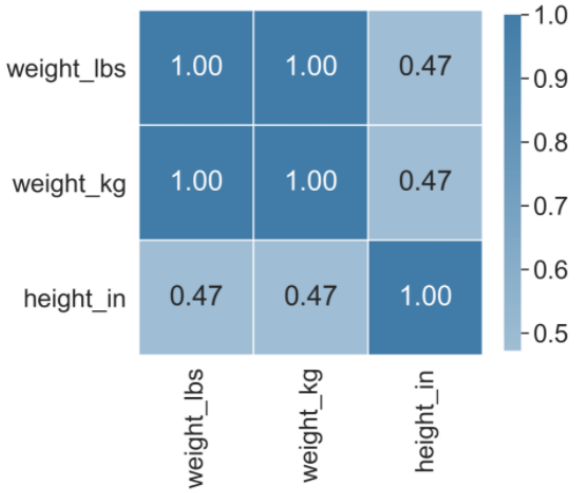
Correlation matrix

```
weights_df.corr()
```

	weight_lbs	weight_kg	height_in
weight_lbs	1.00	1.00	0.47
weight_kg	1.00	1.00	0.47
height_in	0.47	0.47	1.00

Visualizing the correlation matrix

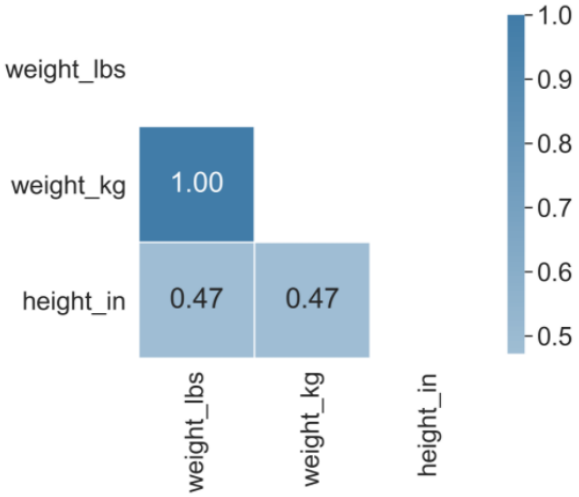
```
cmap = sns.diverging_palette(h_neg=10,
                             h_pos=240,
                             as_cmap=True)

sns.heatmap(weights_df.corr(), center=0,
            cmap=cmap, linewidths=1,
            annot=True, fmt=".2f")
```

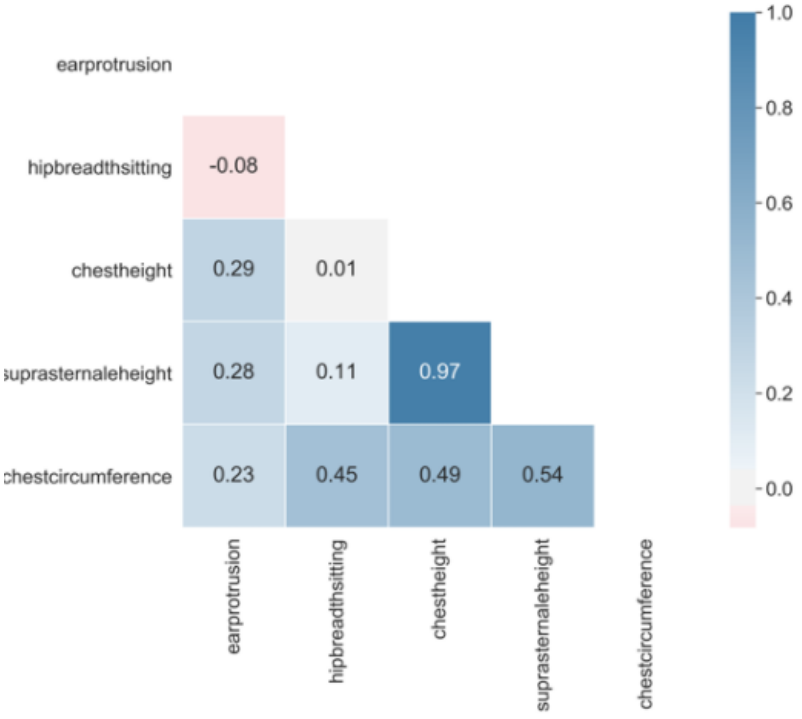


```
corr = weights_df.corr()

mask = np.triu(np.ones_like(corr, dtype=bool))
```

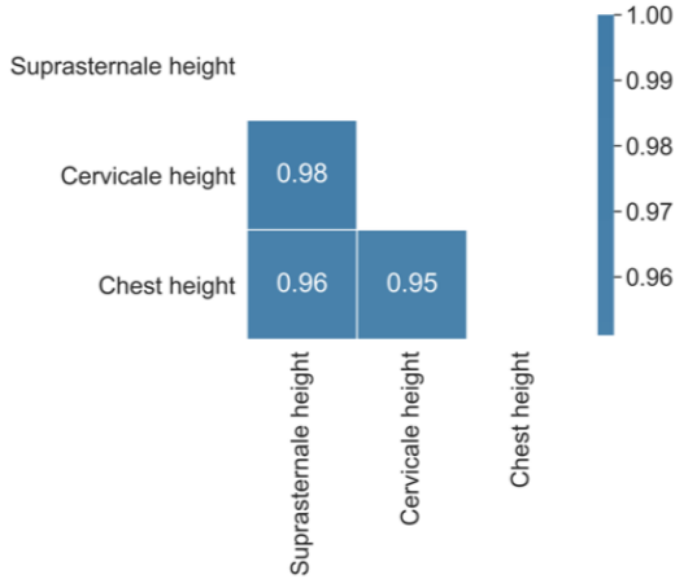
```
array([[ True,  True,  True],
       [False,  True,  True],
       [False, False,  True]])
```

```
sns.heatmap(weights_df.corr(), mask=mask,
            center=0, cmap=cmap, linewidths=1,
            annot=True, fmt=".2f")
```





Removing highly correlated features
→ Avoid small model from overfitting



Removing highly correlated features

```
# Create positive correlation matrix
corr_df = chest_df.corr().abs()
# Create and apply mask
mask = np.triu(np.ones_like(corr_df, dtype=bool))
tri_df = corr_matrix.mask(mask)

tri_df
```

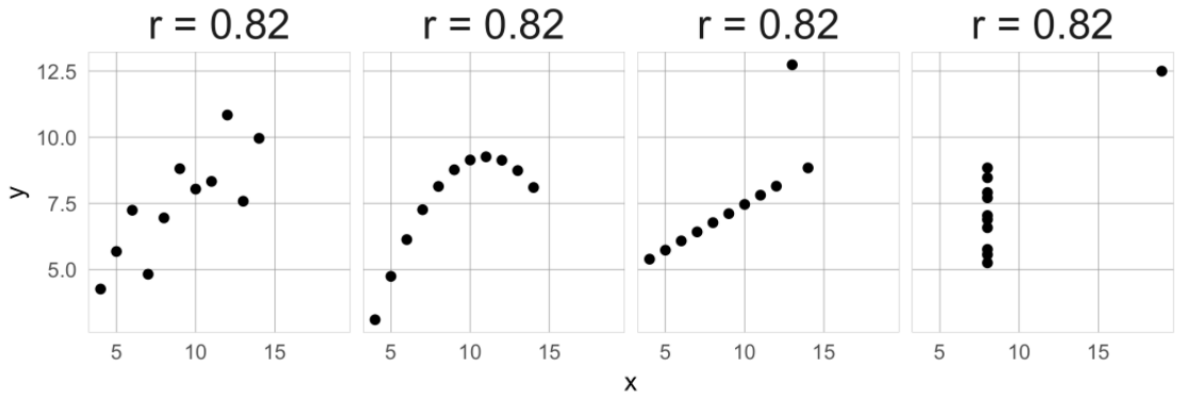
	Suprasternale height	Cervicale height	Chest height
Suprasternale height	NaN	NaN	NaN
Cervicale height	0.983033	NaN	NaN
Chest height	0.956111	0.951101	NaN

```
# Find columns that meet treshold
to_drop = [c for c in tri_df.columns if any(tri_df[c] > 0.95)]

print(to_drop)
```

```
['Suprasternale height', 'Cervicale height']
```

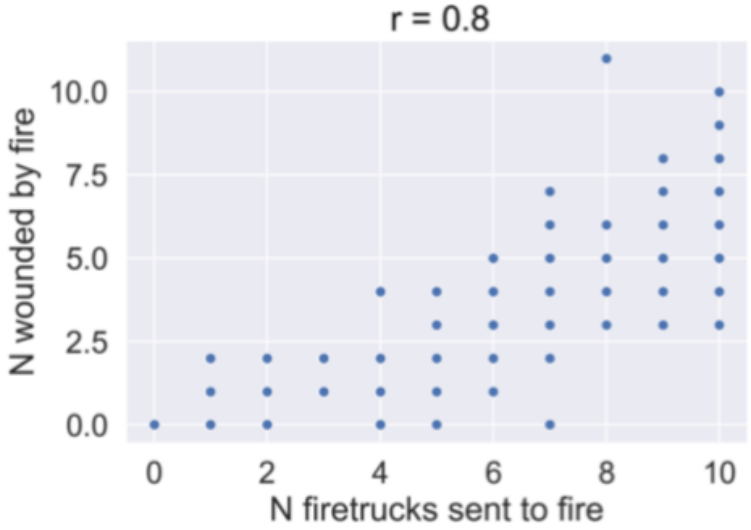
```
# Drop those columns
reduced_df = chest_df.drop(to_drop, axis=1)
```

Correlation caveats - Anscombe's quartet



Correlation caveats - causation

```
sns.scatterplot(x="N firetrucks sent to fire",
                y="N wounded by fire",data=fire_df)
```




```
# Calculate the correlation matrix and take the absolute value
corr_matrix = ansur_df.corr().abs()
# Create a True/False mask and apply it
mask = np.triu(np.ones_like(corr_matrix, dtype=bool))
tri_df = corr_matrix.mask(mask)
# List column names of highly correlated features (r > 0.95)
to_drop = [c for c in tri_df.columns if any(tri_df[c] >  0.95)]
# Drop the features in the to_drop list
reduced_df = ansur_df.drop(to_drop, axis=1)
print("The reduced dataframe has {} columns.".format(reduced_df.shape[1]))
```

Filtering out highly correlated features: You'll calculate the correlation matrix and filter out columns that have a correlation coefficient of more than 0.95 or less than -0.95.

The original dataframe has 99 columns.

<script.py> output:

The reduced dataframe has 88 columns

Ansur dataset sample

Gender	chestdepth	handlength	neckcircumference	shoulderlength	earlength
Female	243	176	326	136	62
Female	219	177	325	135	58
Male	259	193	400	145	71
Male	253	195	380	141	62

Pre-processing the data

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_train_std = scaler.fit_transform(X_train)
```

Creating a logistic regression model

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

lr = LogisticRegression()
lr.fit(X_train_std, y_train)

X_test_std = scaler.transform(X_test)

y_pred = lr.predict(X_test_std)
print(accuracy_score(y_test, y_pred))
```

```
0.99
```

Inspecting the feature coefficients

```
print(lr.coef_)
```

```
array([[-3.  ,  0.14,  7.46,  1.22,  0.87]])
```

```
print(dict(zip(X.columns, abs(lr.coef_[0]))))
```

```
{'chestdepth': 3.0,
 'handlength': 0.14,
 'neckcircumference': 7.46,
 'shoulderlength': 1.22,
 'earlength': 0.87}
```

Features that contribute little to a model

```
X.drop('handlength', axis=1, inplace=True)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

lr.fit(scaler.fit_transform(X_train), y_train)

print(accuracy_score(y_test, lr.predict(scaler.transform(X_test))))
```

```
0.99
```

Recursive Feature Elimination

```
from sklearn.feature_selection import RFE

rfe = RFE(estimator=LogisticRegression(), n_features_to_select=2, verbose=1)
rfe.fit(X_train_std, y_train)
```

```
Fitting estimator with 5 features.
Fitting estimator with 4 features.
Fitting estimator with 3 features.
```

Dropping a feature will affect other feature's coefficients

Inspecting the RFE results

```
X.columns[rfe.support_]
```

```
Index(['chestdepth', 'neckcircumference'], dtype='object')
```

```
print(dict(zip(X.columns, rfe.ranking_)))
```

```
{'chestdepth': 1,
 'handlength': 4,
 'neckcircumference': 1,
 'shoulderlength': 2,
 'earlength': 3}
```

```
print(accuracy_score(y_test, rfe.predict(X_test_std)))
```

```
0.99
```

```
# Fit the scaler on the training features and transform these in one go
X_train_std = scaler.fit_transform(X_train)
# Fit the logistic regression model on the scaled training data
lr.fit(X_train_std, y_train)
# Scale the test features
X_test_std = scaler.transform(X_test)
# Predict diabetes presence on the scaled test set
y_pred = lr.predict(X_test_std)
# Prints accuracy metrics and feature coefficients
print("{0:.1%} accuracy on test set.".format(accuracy_score(y_test, y_pred)))
print(dict(zip(X.columns, abs(lr.coef_[0]).round(2))))
```

Building a diabetes classifier

<script.py> output:
    79.6% accuracy on test set.
    {'pregnant': 0.04, 'glucose': 1.23, 'diastolic': 0.03, 'triceps': 0.24, 'insulin': 0.19, 'bmi': 0.38, 'family': 0.34, 'age': 0.34}

We get almost 80% accuracy on the test set. Take a look at the differences in model coefficients for the different features.

→ removing the low coefficient variables will decrease the accuracy by a little

```
# Create the RFE with a LogisticRegression estimator and 3 features to select
rfe = RFE(estimator=LogisticRegression(), n_features_to_select=3, verbose=1)
# Fits the eliminator to the data
rfe.fit(X_train, y_train)
# Print the features and their ranking (high = dropped early on)
print(dict(zip(X.columns, rfe.ranking_)))
# Print the features that are not eliminated
print(X.columns[rfe.support_])
# Calculates the test set accuracy
acc = accuracy_score(y_test, rfe.predict(X_test))
print("{0:.1%} accuracy on test set.".format(acc))
```

Automatic Recursive Feature Elimination

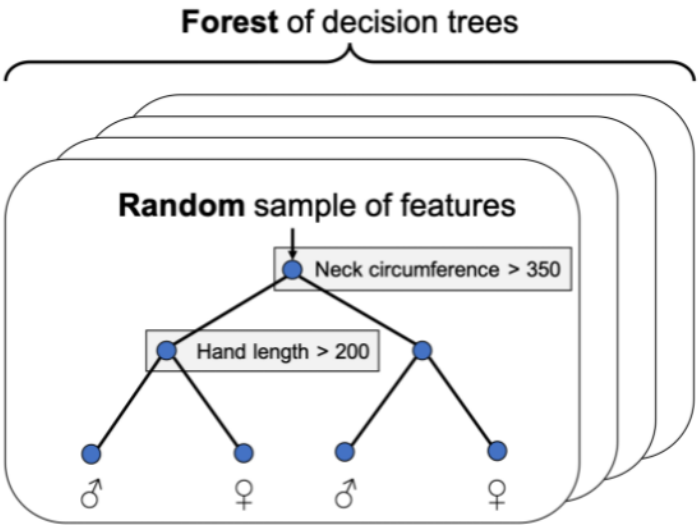
<script.py> output:
Fitting estimator with 8 features.
Fitting estimator with 7 features.
Fitting estimator with 6 features.
Fitting estimator with 5 features.
Fitting estimator with 4 features.
{'pregnant': 5, 'glucose': 1, 'diastolic': 6, 'triceps': 3, 'insulin': 4, 'bmi': 1, 'family': 2, 'age': 1}
Index(['glucose', 'bmi', 'age'], dtype='object')
80.6% accuracy on test set.

Tree-based feature selection

Random forest classifier

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

rf = RandomForestClassifier()

rf.fit(X_train, y_train)

print(accuracy_score(y_test, rf.predict(X_test)))
```

```
0.99
```



Feature importance values

```
rf = RandomForestClassifier()

rf.fit(X_train, y_train)

print(rf.feature_importances_)
```

```
array([0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.04, 0.  , 0.01, 0.01,
       0.  , 0.  , 0.  , 0.  , 0.01, 0.01, 0.  , 0.  , 0.  , 0.  , 0.05,
       ...
       0.  , 0.14, 0.  , 0.  , 0.  , 0.06, 0.  , 0.  , 0.  , 0.  , 0.  ,
       0.  , 0.07, 0.  , 0.  , 0.01, 0.  ])
```

```
print(sum(rf.feature_importances_))
```

```
1.0
```

Feature importance as a feature selector

```
mask = rf.feature_importances_ > 0.1

print(mask)
```

```
array([False, False, ..., True, False])
```

```
X_reduced = X.loc[:, mask]

print(X_reduced.columns)
```

```
Index(['chestheight', 'neckcircumference', 'neckcircumferencebase',
       'shouldercircumference'], dtype='object')
```

RFE with random forests

```
from sklearn.feature_selection import RFE

rfe = RFE(estimator=RandomForestClassifier(),
          n_features_to_select=6, verbose=1)

rfe.fit(X_train,y_train)
```

```
Fitting estimator with 94 features.
Fitting estimator with 93 features
...
Fitting estimator with 8 features.
Fitting estimator with 7 features.
```

```
print(accuracy_score(y_test, rfe.predict(X_test))
```

```
0.99
```

```
from sklearn.feature_selection import RFE

rfe = RFE(estimator=RandomForestClassifier(),
          n_features_to_select=6, step=10, verbose=1)

rfe.fit(X_train,y_train)
```

```
Fitting estimator with 94 features.
Fitting estimator with 84 features.
...
Fitting estimator with 24 features.
Fitting estimator with 14 features.
```

```
print(X.columns[rfe.support_])
```

```
Index(['biacromialbreadth', 'handbreadth', 'handcircumference',
       'neckcircumference', 'neckcircumferencebase', 'shouldercircumference'], dtype='object')
```

```
# Perform a 75% training and 25% test data split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)
# Fit the random forest model to the training data
rf = RandomForestClassifier(random_state=0)
rf.fit(X_train, y_train)
# Calculate the accuracy
acc = accuracy_score(y_test, rf.predict(X_test))
# Print the importances per feature
print(dict(zip(X.columns, rf.feature_importances_.round(2))))
# Print accuracy
print("{0:.1%} accuracy on test set.".format(acc))
```

Building a random forest model

<script.py> output:

{'pregnant': 0.09, 'glucose': 0.21, 'diastolic': 0.08, 'triceps': 0.11, 'insulin': 0.13, 'bmi': 0.09, 'family': 0.12, 'age': 0.16}

77.6% accuracy on test set.

```
# Create a mask for features importances above the threshold
mask = rf.feature_importances_ > 0.15
# Apply the mask to the feature dataset X
reduced_X = X.loc[:, mask]
# prints out the selected column names
print(reduced_X.columns)
```

Random forest for feature selection

<script.py> output:

Index(['glucose', 'age'], dtype='object')

```
# Set the feature eliminator to remove 2 features on each step
rfe = RFE(estimator=RandomForestClassifier(), n_features_to_select=2, step=2, verbose=1)
# Fit the model to the training data
rfe.fit(X_train, y_train)
# Create a mask
mask = rfe.support_
# Apply the mask to the feature dataset X and print the result
reduced_X = X.loc[:, mask]
print(reduced_X.columns)
```

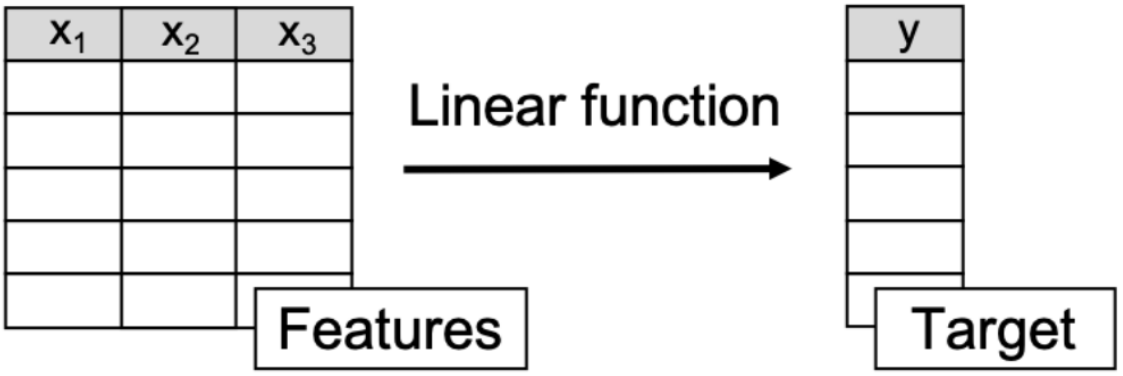
Recursive Feature Elimination with random forests

<script.py> output:

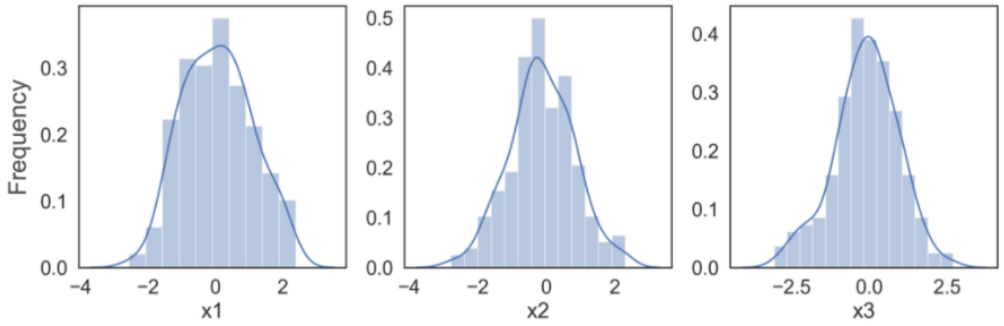
Fitting estimator with 8 features.
Fitting estimator with 6 features.
Fitting estimator with 4 features.
Index(['glucose', 'insulin'], dtype='object')

Regularized linear regression

Linear model concept



Creating our own dataset



Creating our own target feature:

$$y = 20 + 5x_1 + 2x_2 + 0x_3 + error$$

x1	x2	x3
1.76	-0.37	-0.60
0.40	-0.24	-1.12
0.98	1.10	0.77
...	...	...

Linear regression in Python

```
from sklearn.linear_model import LinearRegression

lr = LinearRegression()
lr.fit(X_train, y_train)

# Actual coefficients = [5 2 0]
print(lr.coef_)
```

```
[ 4.95  1.83 -0.05]
```

```
# Actual intercept = 20
print(lr.intercept_)
```

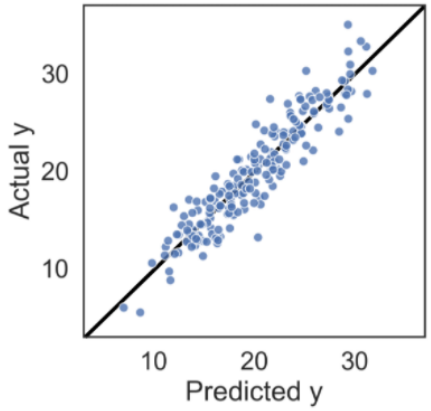
```
19.8
```

```
# Calculates R-squared
print(lr.score(X_test, y_test))
```

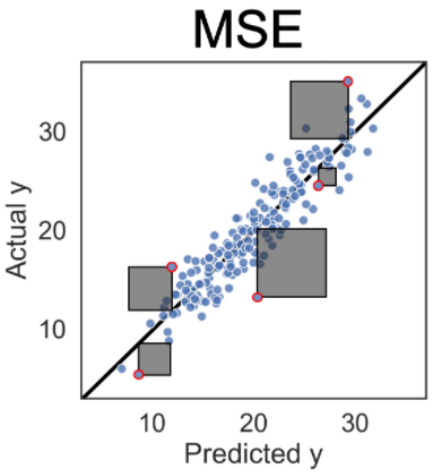
```
0.976
```


```
from sklearn.linear_model import LinearRegression

lr = LinearRegression()
lr.fit(X_train, y_train)

# Actual coefficients = [5 2 0]
print(lr.coef_)
```

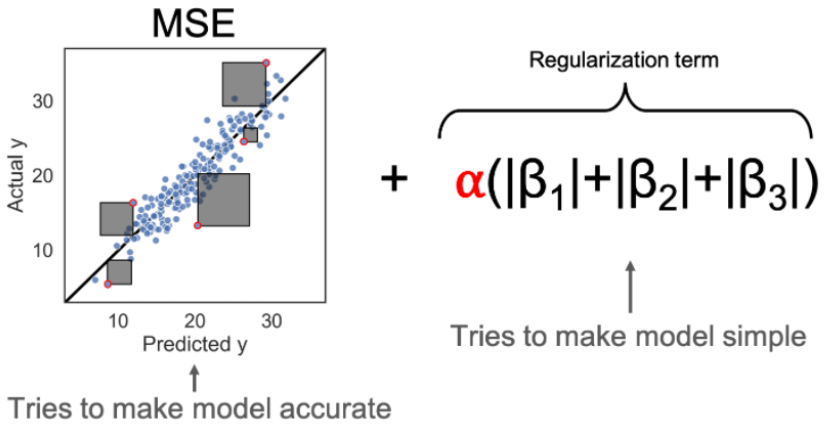
```
[ 4.95  1.83 -0.05]
```

Loss function: Mean Squared Error



Adding regularization



[1] alpha, when it's too low the model might overfit, when it's too high the model might become too simple and inaccurate. One linear model that includes this type of regularization is called Lasso, for least absolute shrinkage

Lasso regressor

```
from sklearn.linear_model import Lasso

la = Lasso()
la.fit(X_train, y_train)

# Actual coefficients = [5 2 0]
print(la.coef_)
```

```
[4.07 0.59 0.  ]
```

```
print(la.score(X_test, y_test))
```

```
0.861
```

```
from sklearn.linear_model import Lasso

la = Lasso(alpha=0.05)
la.fit(X_train, y_train)

# Actual coefficients = [5 2 0]
print(la.coef_)
```

```
[ 4.91  1.76 0.  ]
```

```
print(la.score(X_test, y_test))
```

```
0.974
```

```
# Transform the test set with the pre-fitted scaler
X_test_std = scaler.transform(X_test)
# Calculate the coefficient of determination (R squared) on X_test_std
r_squared = la.score(X_test_std, y_test)
print("The model can predict {0:.1%} of the variance in the test set.".format(r_squared))
# Create a list that has True values when coefficients equal 0
zero_coef = la.coef_ == 0
# Calculate how many features have a zero coefficient
n_ignored = sum(zero_coef)
print("The model has ignored {} out of {} features.".format(n_ignored, len(la.coef_)))
```

```
# Find the highest alpha value with R-squared above 98%
la = Lasso(alpha=0.1, random_state=0)
# Fits the model and calculates performance stats
la.fit(X_train_std, y_train)
r_squared = la.score(X_test_std, y_test)
n_ignored_features = sum(la.coef_ == 0)
# Print peformance stats
print("The model can predict {0:.1%} of the variance in the test set.".format(r_squared))
print("{} out of {} features were ignored.".format(n_ignored_features, len(la.coef_)))
```

<script.py> output:

The model can predict 98.3% of the variance in the test set.

64 out of 91 features were ignored.

Combining feature selectors

Lasso regressor

```
from sklearn.linear_model import Lasso

la = Lasso(alpha=0.05)
la.fit(X_train, y_train)

# Actual coefficients = [5 2 0]
print(la.coef_)
```

```
[ 4.91  1.76 0. ]
```

```
print(la.score(X_test, y_test))
```

```
0.974
```

LassoCV regressor

```
from sklearn.linear_model import LassoCV

lcv = LassoCV()

lcv.fit(X_train, y_train)

print(lcv.alpha_)
```

```
0.09
```

```
mask = lcv.coef_ != 0

print(mask)
```

```
[ True True False ]
```

```
reduced_X = X.loc[:, mask]
```

Feature selection with LassoCV

```
from sklearn.linear_model import LassoCV

lcv = LassoCV()
lcv.fit(X_train, y_train)

lcv.score(X_test, y_test)
```

```
0.99
```

```
lcv_mask = lcv.coef_ != 0
sum(lcv_mask)
```

```
66
```

Feature selection with random forest

```
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestRegressor

rfe_rf = RFE(estimator=RandomForestRegressor(),
             n_features_to_select=66, step=5, verbose=1)

rfe_rf.fit(X_train, y_train)
rf_mask = rfe_rf.support_
```

Feature selection with gradient boosting

```
from sklearn.feature_selection import RFE
from sklearn.ensemble import GradientBoostingRegressor

rfe_gb = RFE(estimator=GradientBoostingRegressor(),
             n_features_to_select=66, step=5, verbose=1)

rfe_gb.fit(X_train, y_train)
gb_mask = rfe_gb.support_
```

Combining the feature selectors

```
import numpy as np

votes = np.sum([lcv_mask, rf_mask, gb_mask], axis=0)

print(votes)
```

```
array([3, 2, 2, ..., 3, 0, 1])
```

```
mask = votes >= 2
reduced_X = X.loc[:, mask]
```

Creating a LassoCV regressor

```
from sklearn.linear_model import LassoCV
# Create and fit the LassoCV model on the training set
lcv = LassoCV()
lcv.fit(X_train, y_train)
print('Optimal alpha = {0:.3f}'.format(lcv.alpha_))
# Calculate R squared on the test set
r_squared = lcv.score(X_test, y_test)
print('The model explains {0:.1%} of the test set variance'.format(r_squared))
# Create a mask for coefficients not equal to zero
lcv_mask = lcv.coef_ != 0
print('{} features out of {} selected'.format(sum(lcv_mask), len(lcv_mask)))
```

<script.py> output:

Optimal alpha = 0.089

The model explains 88.2% of the test set variance

26 features out of 32 selected

Combining 3 feature selectors

```
# Sum the votes of the three models
votes = np.sum([lcv_mask, rf_mask, gb_mask], axis=0)
# Create a mask for features selected by all 3 models
meta_mask = votes >= 3
# Apply the dimensionality reduction on X
X_reduced = X.loc[:, meta_mask]
# Plug the reduced dataset into a linear regression pipeline
X_train, X_test, y_train, y_test = train_test_split(X_reduced, y, test_size=0.3, random_state=0)
lm.fit(scaler.fit_transform(X_train), y_train)
r_squared = lm.score(scaler.transform(X_test), y_test)
print('The model can explain {0:.1%} of the variance in the test set using {1:} features.'.format(r_squared,
 len(lm.coef_)))
```

<script.py> output:

The model can explain 86.8% of the variance in the test set using 7 features.

Feature extraction