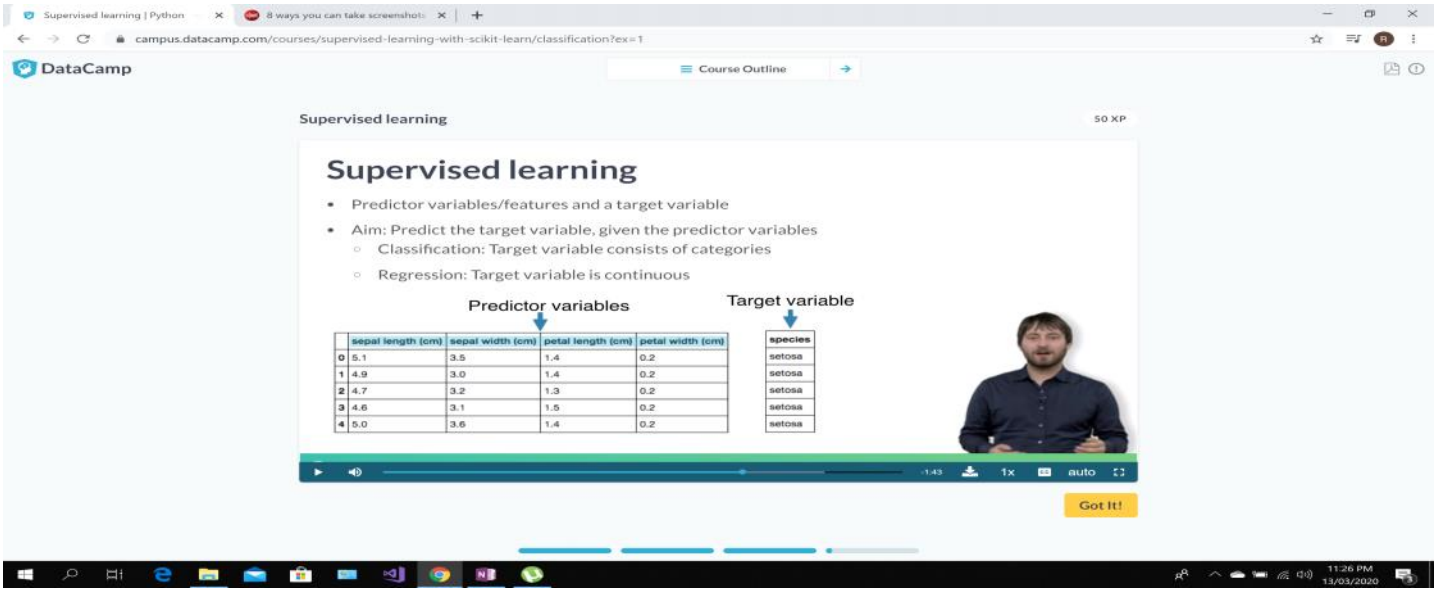
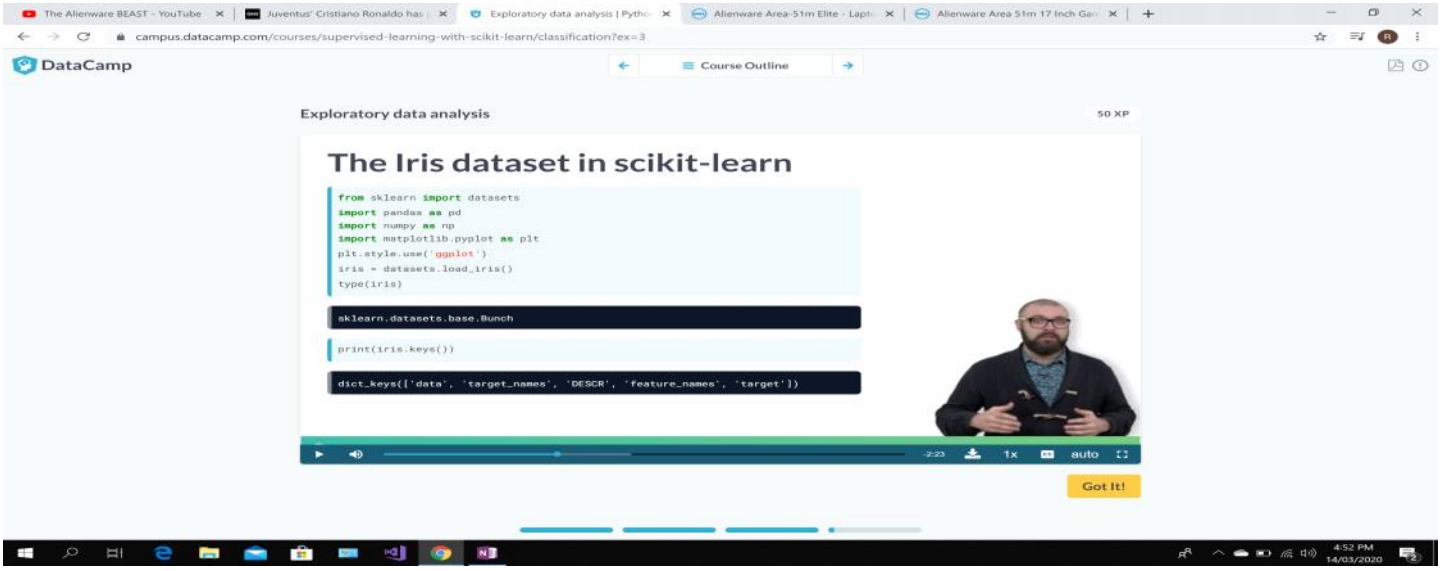
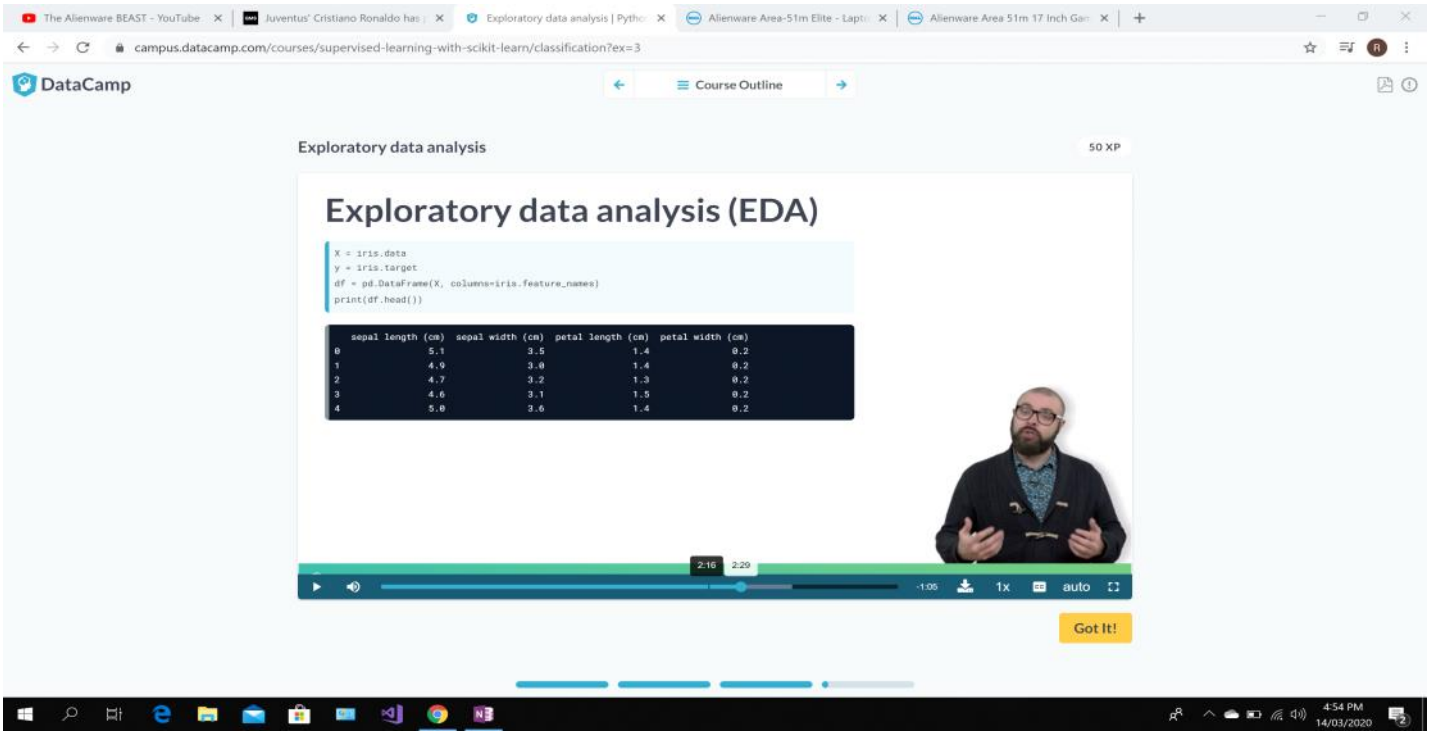
Supervised learning

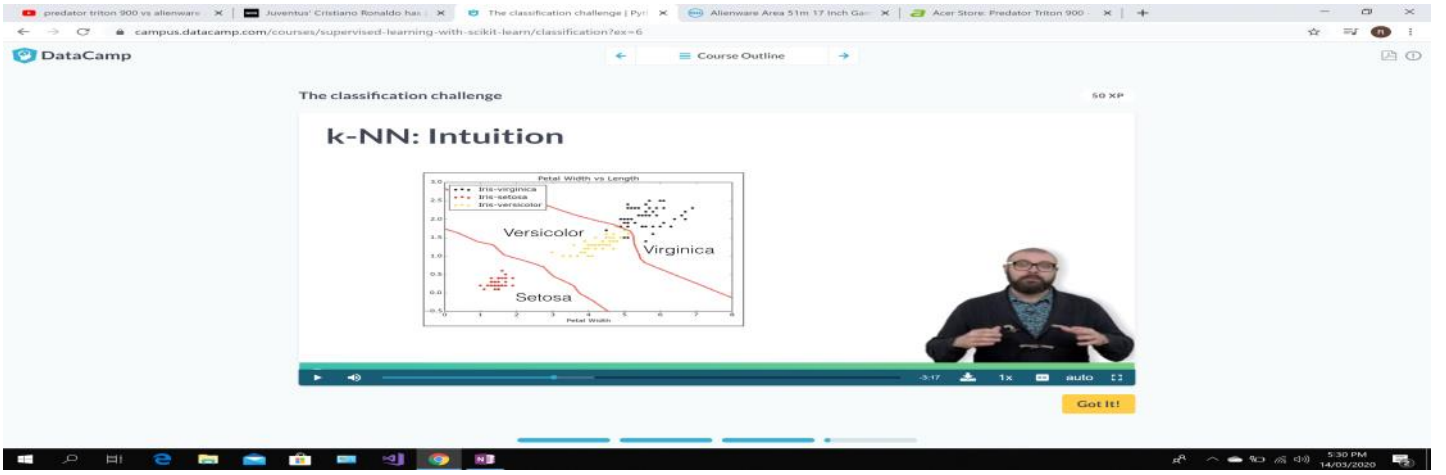
Friday, 13 March 2020 11:18 PM

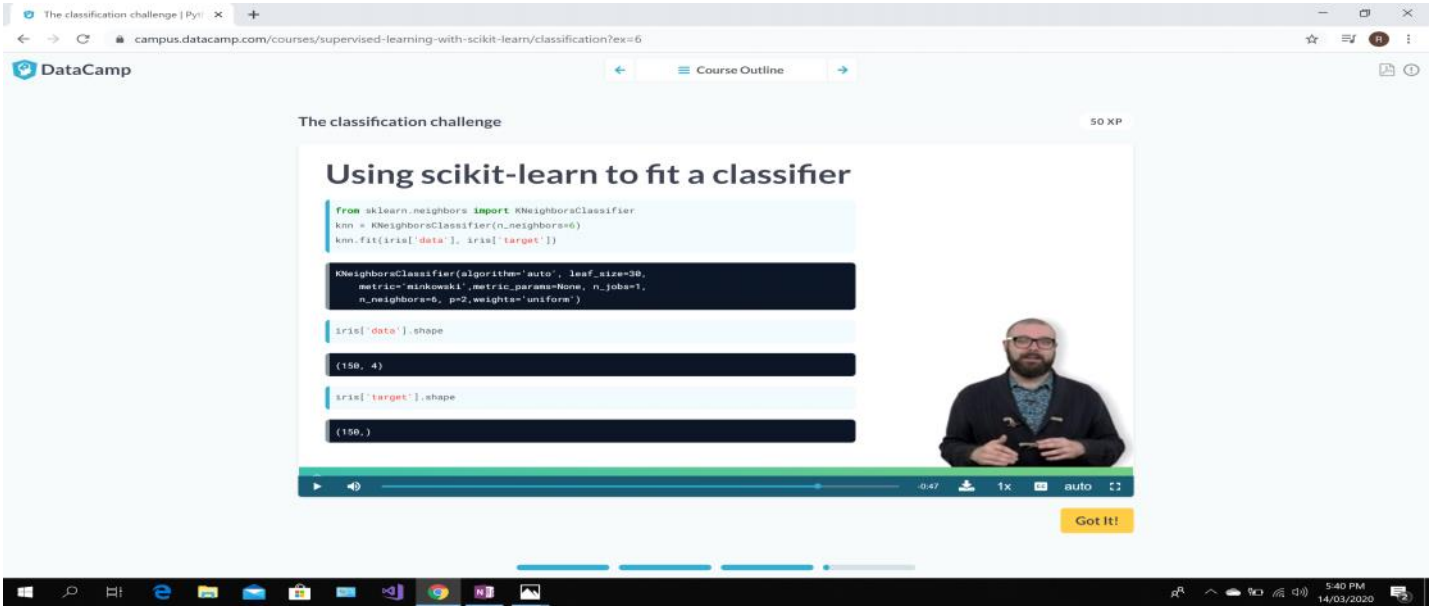
CLASSIFICATION





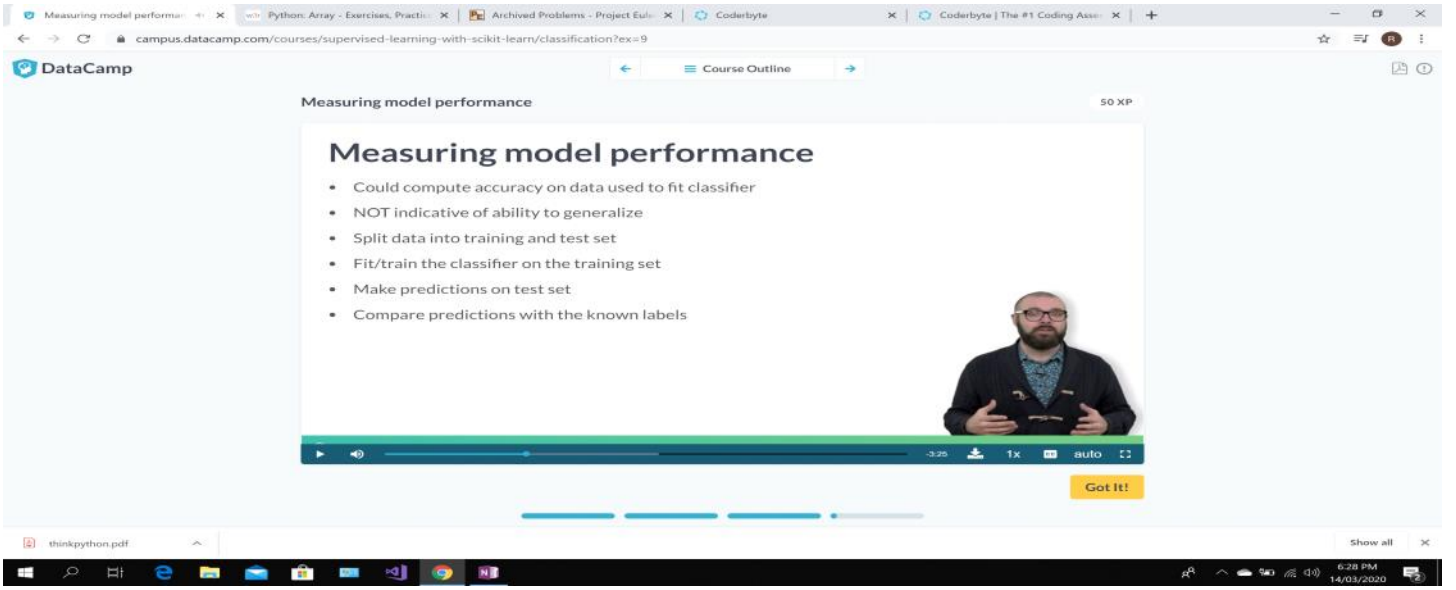

DataCamp
Course Outline
Exploratory data analysis
50 XP
Exploratory data analysis (EDA)
X = iris.data
y = iris.target
df = pd.DataFrame(X, columns=iris.feature_names)
print(df.head())
sepal length (cm) sepal width (cm) petal length (cm) petal width (cm)
0 5.1 3.5 1.4 0.2
1 4.9 3.0 1.4 0.2
2 4.7 3.2 1.3 0.2
3 4.6 3.1 1.5 0.2
4 5.0 3.6 1.4 0.2
2:16
2:29
-1:05
1x
auto
Got It!
4:54 PM
14/03/2020

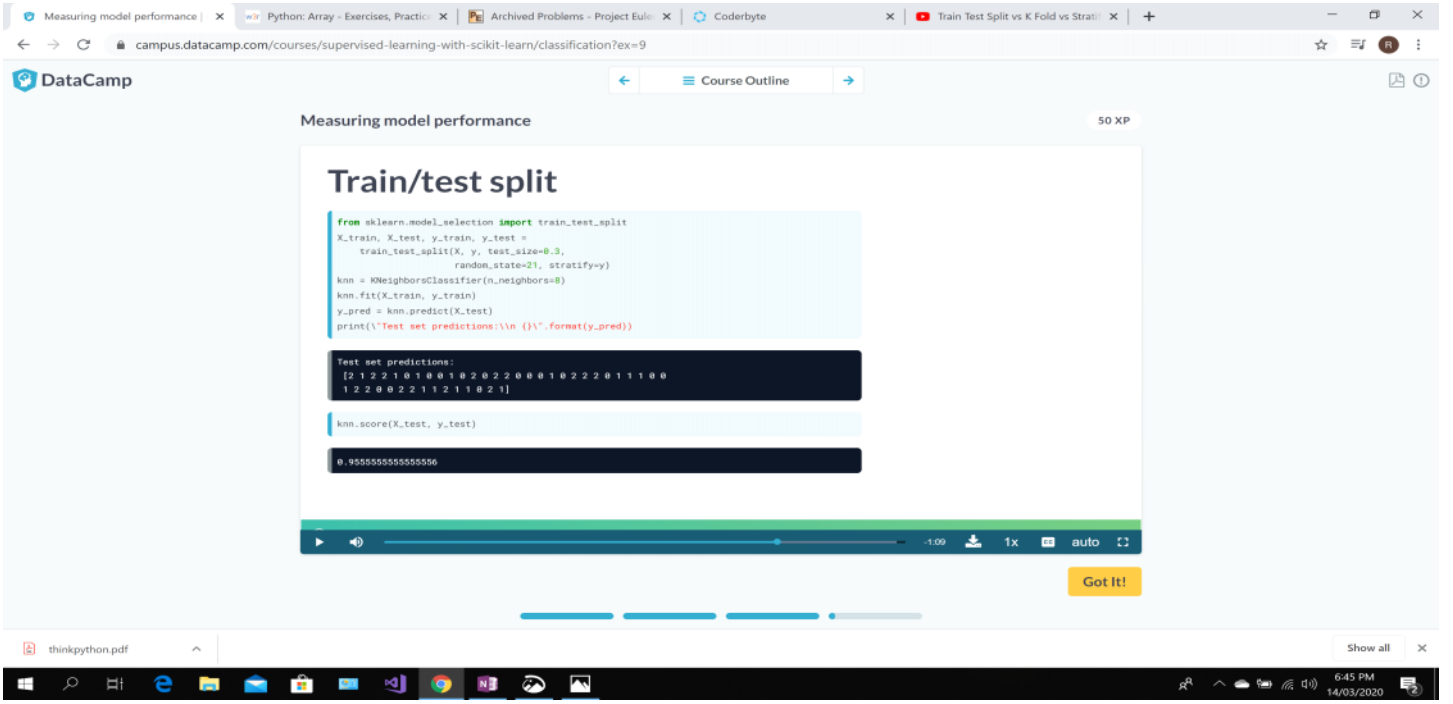

DataCamp
Course Outline
The classification challenge
50 XP
k-NN: Intuition
Petal Width vs Length
Iris-virginica
Iris-setosa
Iris-versicolor
Versicolor
Virginica
Setosa
Petal Width
-3:17
1x
auto
Got It!
5:30 PM
14/03/2020

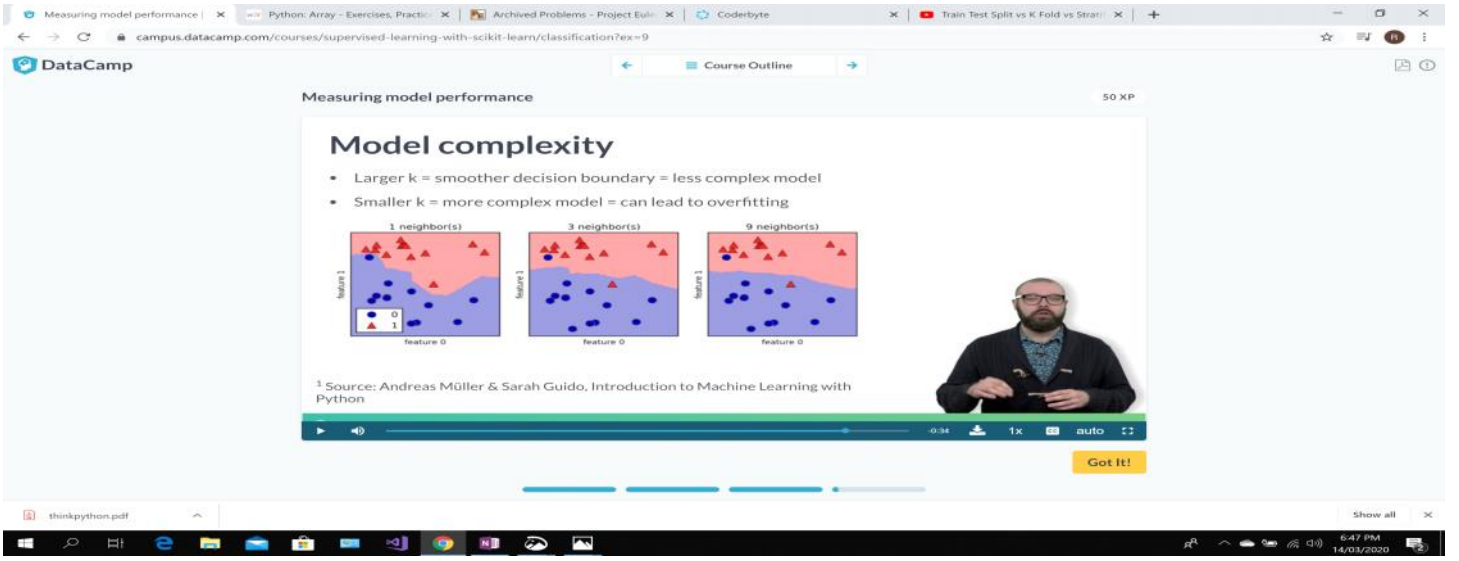

DataCamp
Course Outline
The classification challenge
50 XP
Using scikit-learn to fit a classifier
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(iris['data'], iris['target'])
KNeighborsClassifier(algorithm='auto', leaf_size=30,
metric='minkowski',metric_params=None, n_jobs=1,
n_neighbors=6, p=2,weights='uniform')
iris['data'].shape
(150, 4)
iris['target'].shape
(150,)
-0:47
1x
auto
Got It!
5:40 PM
14/03/2020

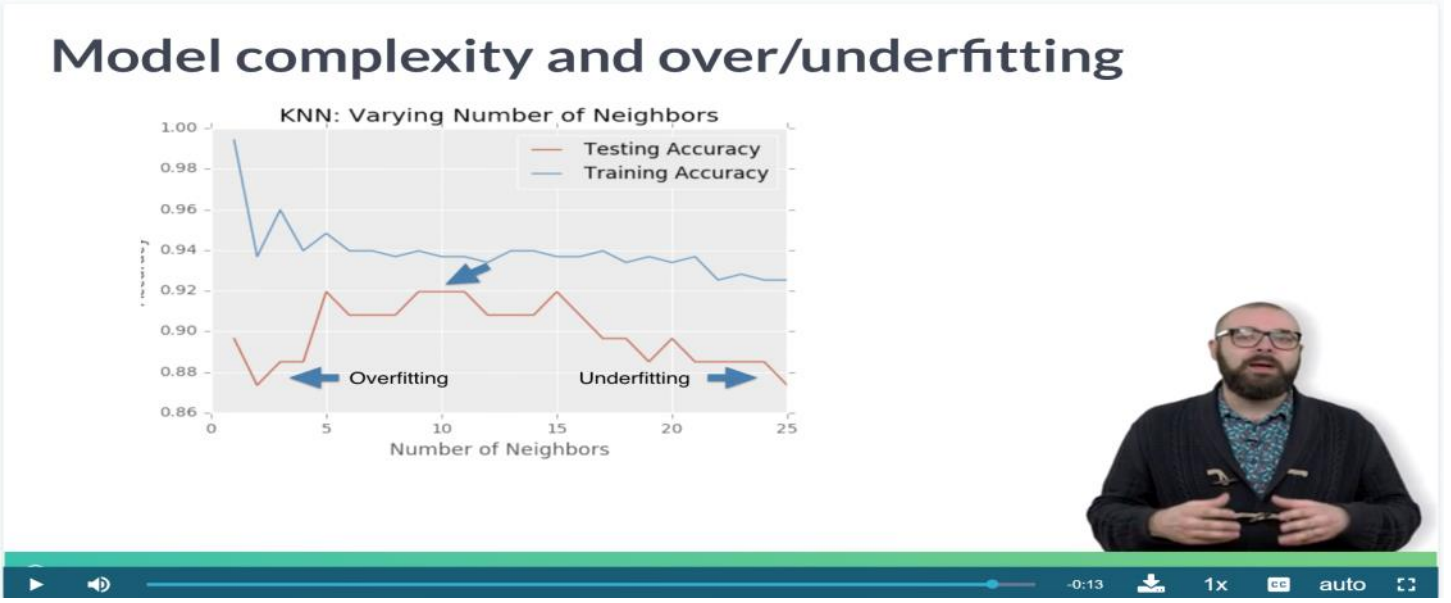
Points to be noted:

- The features and targets should be passed as numpy array or pandas dataframe
- It also requires that features should take continuous values
- It also requires no missing data






DataCamp
Course Outline
Measuring model performance
50 XP
Model complexity
Larger k = smoother decision boundary = less complex model
Smaller k = more complex model = can lead to overfitting
1 neighbor(s)
3 neighbor(s)
9 neighbor(s)
feature 0
feature 1
Source: Andreas Müller & Sarah Guido, Introduction to Machine Learning with Python
Got It!


Model complexity and over/underfitting
KNN: Varying Number of Neighbors
Testing Accuracy
Training Accuracy
Overfitting
Underfitting
Number of Neighbors
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0
5
10
15
20
25
-0:13
1x
auto

REGRESSION


Creating feature and target arrays
X = boston.drop('MEDV', axis=1).values
y = boston['MEDV'].values

Predicting house value from a single feature

```
X_rooms = X[:,5]
type(X_rooms), type(y)
```

```
(numpy.ndarray, numpy.ndarray)
```

```
y = y.reshape(-1, 1)
X_rooms = X_rooms.reshape(-1, 1)
```

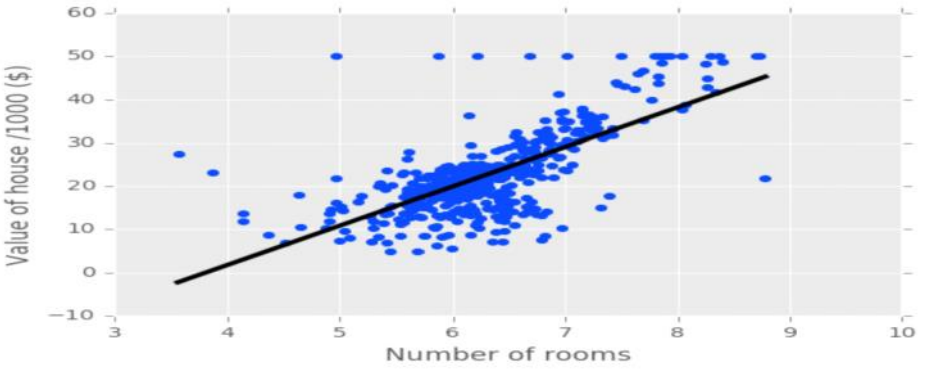
2:05

Fitting a regression model

```
import numpy as np
from sklearn.linear_model import LinearRegression

reg = LinearRegression()
reg.fit(X_rooms, y)
prediction_space = np.linspace(min(X_rooms),
                               max(X_rooms)).reshape(-1, 1)
```

```
plt.scatter(X_rooms, y, color='blue')
plt.plot(prediction_space, reg.predict(prediction_space),
             color='black', linewidth=3)
plt.show()
```

Fitting a regression model



Regression mechanics

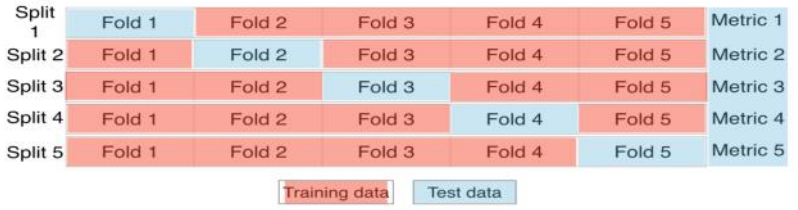
- y = ax + b
  - y = target
  - x = single feature
  - a, b = parameters of model
- How do we choose a and b?
- Define an error functions for any given line
  - Choose the line that minimizes the error function

Linear regression on all features

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size = 0.3, random_state=42)
reg_all = LinearRegression()
reg_all.fit(X_train, y_train)
y_pred = reg_all.predict(X_test)
reg_all.score(X_test, y_test)
```

```
0.71122600574849526
```

Cross-validation basics



Cross-validation in scikit-learn

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression
reg = LinearRegression()
cv_results = cross_val_score(reg, X, y, cv=5)
print(cv_results)
```

```
[ 0.63919994  0.71386698  0.58702344  0.07923081 -0.25294154]
```

```
np.mean(cv_results)
```

```
0.35327592439587058
```

Why regularize?

- Recall: Linear regression minimizes a loss function
- It chooses a coefficient for each feature variable
- Large coefficients can lead to overfitting
- Penalizing large coefficients: Regularization

Ridge regression

- Loss function = OLS loss function +

$$\alpha * \sum_{i=1}^{n} {a_i}^2$$

- Alpha: Parameter we need to choose
- Picking alpha here is similar to picking k in k-NN
- Hyperparameter tuning (More in Chapter 3)
- Alpha controls model complexity
  - Alpha = 0: We get back OLS (Can lead to overfitting)
  - Very high alpha: Can lead to underfitting

Ridge regression in scikit-learn

```
from sklearn.linear_model import Ridge
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size = 0.3, random_state=42)
ridge = Ridge(alpha=0.1, normalize=True)
ridge.fit(X_train, y_train)
ridge_pred = ridge.predict(X_test)
ridge.score(X_test, y_test)
```

```
0.69969382751273179
```

Lasso regression

- Loss function = OLS loss function +

$$\alpha * \sum_{i=1}^{n} |a_i|$$

Lasso regression in scikit-learn

```
from sklearn.linear_model import Lasso
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size = 0.3, random_state=42)
lasso = Lasso(alpha=0.1, normalize=True)
lasso.fit(X_train, y_train)
lasso_pred = lasso.predict(X_test)
lasso.score(X_test, y_test)
```

```
0.59502295353285506
```

Lasso regression for feature selection

- Can be used to select important features of a dataset
- Shrinks the coefficients of less important features to exactly 0

Lasso for feature selection in scikit-learn

```
from sklearn.linear_model import Lasso
names = boston.drop('MEDV', axis=1).columns
lasso = Lasso(alpha=0.1)
lasso_coef = lasso.fit(X, y).coef_
_ = plt.plot(range(len(names)), lasso_coef)
_ = plt.xticks(range(len(names)), names, rotation=60)
_ = plt.ylabel('Coefficients')
plt.show()
```

FINE TUNING YOUR MODEL

Class imbalance example: Emails

- Spam classification
  - 99% of emails are real; 1% of emails are spam
- Could build a classifier that predicts ALL emails as real
  - 99% accurate!
  - But horrible at actually classifying spam
  - Fails at its original purpose
- Need more nuanced metrics

Diagnosing classification predictions

- Confusion matrix

	Predicted: Spam Email	Predicted: Real Email
Actual: Spam Email	True Positive	False Negative
Actual: Real Email	False Positive	True Negative

- Accuracy:

$$\frac{tp + tn}{tp + tn + fp + fn}$$

Metrics from the confusion matrix

- Precision $\frac{tp}{tp + fp}$
- Recall $\frac{tp}{tp + fn}$
- F1score: $2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$
- High precision: Not many real emails predicted as spam
- High recall: Predicted most spam emails correctly

Confusion matrix in scikit-learn

```
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
knn = KNeighborsClassifier(n_neighbors=8)
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.4, random_state=42)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
```

Confusion matrix in scikit-learn

```
print(confusion_matrix(y_test, y_pred))
```

```
[[52  7]
 [ 3 112]]
```

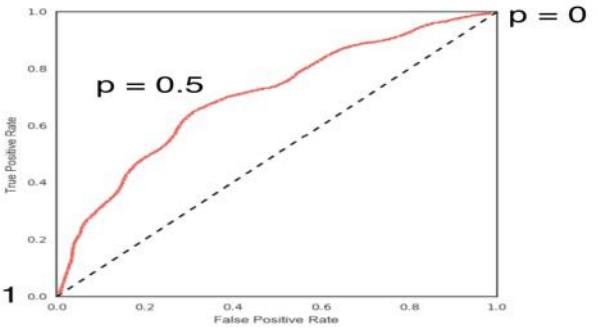
```
print(classification_report(y_test, y_pred))
```

```
             precision    recall  f1-score   support
          0       0.95      0.88      0.91        59
          1       0.94      0.97      0.96       115
avg / total       0.94      0.94      0.94       174
```

Logistic regression in scikit-learn

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
logreg = LogisticRegression()
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.4, random_state=42)
logreg.fit(X_train, y_train)
y_pred = logreg.predict(X_test)
```

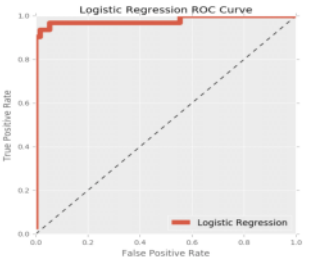
The ROC curve



Plotting the ROC curve

```
from sklearn.metrics import roc_curve
y_pred_prob = logreg.predict_proba(X_test)[:,1]
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr, tpr, label='Logistic Regression')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Logistic Regression ROC Curve')
plt.show();
```

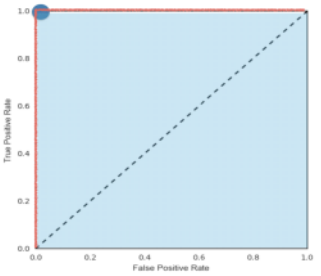
Plotting the ROC curve




```
logreg.predict_proba(X_test)[:,1]
```

Area under the ROC curve (AUC)

- Larger area under the ROC curve = better model



AUC in scikit-learn

```
from sklearn.metrics import roc_auc_score
logreg = LogisticRegression()
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.4, random_state=42)
logreg.fit(X_train, y_train)
y_pred_prob = logreg.predict_proba(X_test)[:,1]
roc_auc_score(y_test, y_pred_prob)
```

```
0.997466216216
```

AUC using cross-validation

```
from sklearn.model_selection import cross_val_score
cv_scores = cross_val_score(logreg, X, y, cv=5,
                            scoring='roc_auc')
print(cv_scores)
```

```
[ 0.99673203  0.99183007  0.99583796  1.          0.96140652]
```

Hyperparameter tuning

- Linear regression: Choosing parameters
- Ridge/lasso regression: Choosing alpha
- k-Nearest Neighbors: Choosing n_neighbors
- Parameters like alpha and k: Hyperparameters
- Hyperparameters cannot be learned by fitting the model

GridSearchCV in scikit-learn

```
from sklearn.model_selection import GridSearchCV
param_grid = {'n_neighbors': np.arange(1, 50)}
knn = KNeighborsClassifier()
knn_cv = GridSearchCV(knn, param_grid, cv=5)
knn_cv.fit(X, y)
knn_cv.best_params_
```

```
{'n_neighbors': 12}
```

```
knn_cv.best_score_
```

```
0.933216168717
```

Hold-out set reasoning

- How well can the model perform on never before seen data?
- Using ALL data for cross-validation is not ideal
- Split data into training and hold-out set at the beginning
- Perform grid search cross-validation on training set
- Choose best hyperparameters and evaluate on hold-out set

PREPROCESSING AND PIPELINES

Dealing with categorical features

- Scikit-learn will not accept categorical features by default
- Need to encode categorical features numerically
- Convert to 'dummy variables'
  - 0: Observation was NOT that category
  - 1: Observation was that category

Dummy variables

Origin
US
Europe
Asia

origin_Asia	origin_Europe	origin_US
0	0	1
0	1	0
1	0	0

Dummy variables

Origin
US
Europe
Asia

origin_Asia	origin_US
0	1
0	0
1	0

Dealing with categorical features in Python

- scikit-learn: OneHotEncoder()
- pandas: get_dummies()

Automobile dataset

- mpg: Target Variable
- Origin: Categorical Feature

	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

Encoding dummy variables

```
import pandas as pd
df = pd.read_csv('auto.csv')
df_origin = pd.get_dummies(df)
print(df_origin.head())
```

```
    mpg  displ   hp  weight  accel  size  origin_Asia  origin_Europe  \\
0  18.0  250.0   88    3139   14.5  15.0            0              0
1   9.0  304.0  193    4732   18.5  20.0            0              0
2  36.1   91.0   60    1800   16.4  10.0            1              0
3  18.5  250.0   98    3525   19.0  15.0            0              0
4  34.3   97.0   78    2188   15.8  10.0            0              1
   origin_US
0          1
1          1
2          0
3          1
4          0
```

Encoding dummy variables

```
df_origin = df_origin.drop('origin_Asia', axis=1)
print(df_origin.head())
```

```
    mpg  displ   hp  weight  accel  size  origin_Europe  origin_US
0  18.0  250.0   88    3139   14.5  15.0              0          1
1   9.0  304.0  193    4732   18.5  20.0              0          1
2  36.1   91.0   60    1800   16.4  10.0              0          0
3  18.5  250.0   98    3525   19.0  15.0              0          1
4  34.3   97.0   78    2188   15.8  10.0              1          0
```

Linear regression with dummy variables

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.3, random_state=42)
```

```
ridge = Ridge(alpha=0.5, normalize=True).fit(X_train,
                                             y_train)
```

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

```
0.719064519022
```

Dropping missing data

```
df.insulin.replace(0, np.nan, inplace=True)
df.triceps.replace(0, np.nan, inplace=True)
df.bmi.replace(0, np.nan, inplace=True)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
pregnancies    768 non-null int64
glucose        768 non-null int64
diastolic      768 non-null int64
triceps        541 non-null float64
insulin        394 non-null float64
bmi            757 non-null float64
dpf            768 non-null float64
age            768 non-null int64
diabetes       768 non-null int64
dtypes: float64(4), int64(5)
memory usage: 54.1 KB
```

Dropping missing data

```
df = df.dropna()
df.shape
```

```
(393, 9)
```

Imputing missing data

- Making an educated guess about the missing values
- Example: Using the mean of the non-missing entries

```
from sklearn.preprocessing import Imputer
imp = Imputer(missing_values='NaN', strategy='mean', axis=0)
imp.fit(X)
X = imp.transform(X)
```

Imputing within a pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import Imputer
imp = Imputer(missing_values='NaN', strategy='mean', axis=0)
logreg = LogisticRegression()
steps = [('imputation', imp),
            ('logistic_regression', logreg)]
pipeline = Pipeline(steps)
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.3, random_state=42)
```

Imputing within a pipeline

```
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)
pipeline.score(X_test, y_test)
```

```
0.75324675324675328
```

Points to be noted:

- In a pipeline each step but the last must be transformer
- And the last step must be a estimator like regressor or classifier

Why scale your data?

- Many models use some form of distance to inform them
- Features on larger scales can unduly influence the model
- Example: k-NN uses distance explicitly when making predictions
- We want features to be on a similar scale
- Normalizing (or scaling and centering)

Ways to normalize your data

- Standardization: Subtract the mean and divide by variance
- All features are centered around zero and have variance one
- Can also subtract the minimum and divide by the range
- Minimum zero and maximum one
- Can also normalize so the data ranges from -1 to +1
- See scikit-learn docs for further details

Scaling in scikit-learn

```
from sklearn.preprocessing import scale
X_scaled = scale(X)
```

```
np.mean(X), np.std(X)
```

```
(8.13421922452, 16.7265339794)
```

```
np.mean(X_scaled), np.std(X_scaled)
```

```
(2.54662653149e-15, 1.0)
```

Scaling in a pipeline

```
from sklearn.preprocessing import StandardScaler
steps = [('scaler', StandardScaler()),
         ('knn', KNeighborsClassifier())]
pipeline = Pipeline(steps)
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=21)
knn_scaled = pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)
accuracy_score(y_test, y_pred)
```

```
0.956
```

```
knn_unscaled = KNeighborsClassifier().fit(X_train, y_train)
knn_unscaled.score(X_test, y_test)
```

```
0.928
```

CV and scaling in a pipeline

```
steps = [('scaler', StandardScaler()),
        (('knn', KNeighborsClassifier())]
pipeline = Pipeline(steps)
parameters = {knn__n_neighbors: np.arange(1, 50)}
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=21)
cv = GridSearchCV(pipeline, param_grid=parameters)
cv.fit(X_train, y_train)
y_pred = cv.predict(X_test)
```

Scaling and CV in a pipeline

```
print(cv.best_params_)
```

```
{'knn__n_neighbors': 41}
```

```
print(cv.score(X_test, y_test))
```

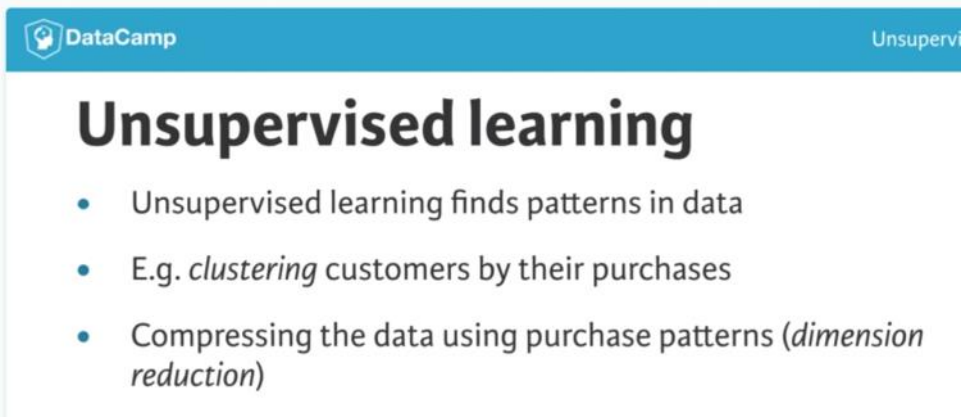
```
0.956
```

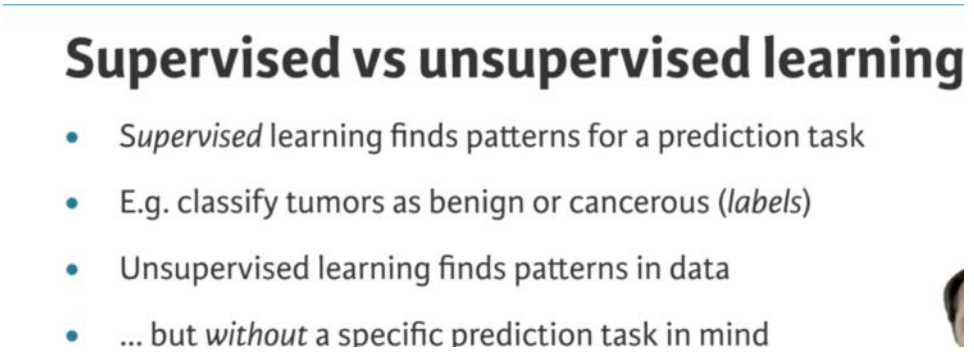
```
print(classification_report(y_test, y_pred))
```

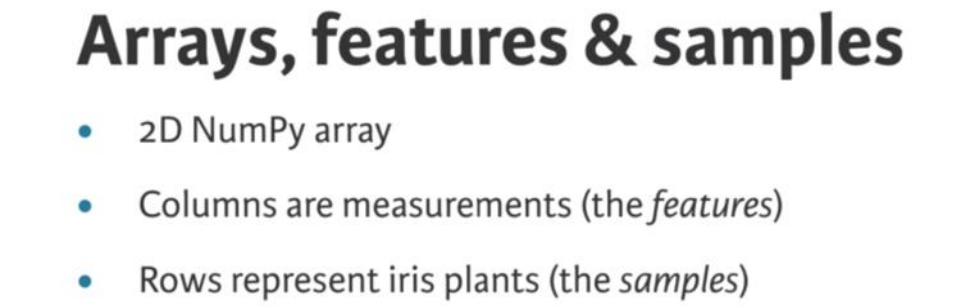
```
             precision    recall  f1-score   support

          0       0.97      0.90      0.93        39
          1       0.95      0.99      0.97        75
avg / total       0.96      0.96      0.96       114
```

Unsupervised learning

Monday, 16 March 2020 4:53 PM







Iris data is 4-dimensional

- Iris samples are points in 4 dimensional space
- Dimension = number of features
- Dimension too high to visualize!
- ... but unsupervised learning gives insight

k-means clustering

- Finds clusters of samples
- Number of clusters must be specified
- Implemented in **sklearn** ("scikit-learn")

```
In [1]: print(samples)
[[ 5.   3.3  1.4  0.2]
 [ 5.   3.5  1.3  0.3]
 [ 4.9  2.4  3.3  1. ]
 [ 6.3  2.8  5.1  1.5]
 ...
 [ 7.2  3.2  6.   1.8]]

In [2]: from sklearn.cluster import KMeans

In [3]: model = KMeans(n_clusters=3)

In [4]: model.fit(samples)
Out[4]: KMeans(algorithm='auto', ...)

In [5]: labels = model.predict(samples)

In [6]: print(labels)
[0 0 1 1 0 1 2 1 0 1 ...]
```

Cluster labels for new samples

- New samples can be assigned to existing clusters
- k-means remembers the mean of each cluster (the "centroids")
- Finds the nearest centroid to each new sample

Cluster labels for new samples

```
In [7]: print(new_samples)
[[ 5.7  4.4  1.5  0.4]
 [ 6.5  3.   5.5  1.8]
 [ 5.8  2.7  5.1  1.9]]

In [8]: new_labels = model.predict(new_samples)

In [9]: print(new_labels)
[0 2 1]
```

Scatter plots

```
In [1]: import matplotlib.pyplot as plt

In [2]: xs = samples[:,0]

In [3]: ys = samples[:,2]

In [4]: plt.scatter(xs, ys, c=labels)
```

Iris: clusters vs species

- k-means found 3 clusters amongst the iris samples
- Do the clusters correspond to the species?

```
species  setosa  versicolor  virginica
labels
0             0           2         36
1            50           0          0
2             0          48         14
```

Cross tabulation with pandas

- Clusters vs species is a "cross-tabulation"
- Use the **pandas** library
- Given the species of each sample as a list **species**

```
In [1]: print(species)
['setosa', 'setosa', 'versicolor', 'virginica', ... ]
```

Aligning labels and species

```
In [2]: import pandas as pd

In [3]: df = pd.DataFrame({'labels': labels, 'species': species})

In [4]: print(df)
    labels     species
0        1      setosa
1        1      setosa
2        2  versicolor
3        2   virginica
4        1      setosa
...
```

Crosstab of labels and species

```
In [5]: ct = pd.crosstab(df['labels'], df['species'])

In [6]: print(ct)
species  setosa  versicolor  virginica
labels
0             0           2         36
1            50           0          0
2             0          48         14
```

How to evaluate a clustering, if there were no species information?

Measuring clustering quality

- Using only samples and their cluster labels
- A good clustering has tight clusters
- ... and samples in each cluster bunched together

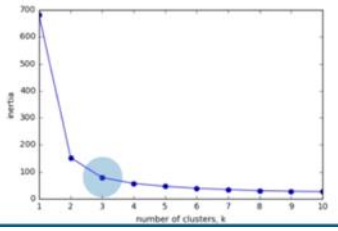
Inertia measures clustering quality

- Measures how spread out the clusters are (*lower* is better)
- Distance from each sample to centroid of its cluster
- After **fit()**, available as attribute **inertia_**
- k-means attempts to minimize the inertia when choosing clusters

```
In [1]: from sklearn.cluster import KMeans

In [2]: model = KMeans(n_clusters=3)

In [3]: model.fit(samples)

In [4]: print(model.inertia_)
78.9408414261
```

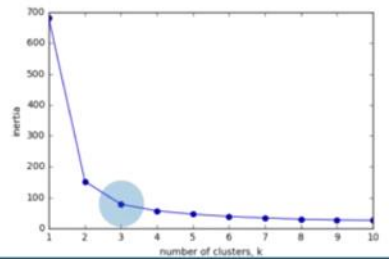
The number of clusters

- Clusterings of the iris dataset with different numbers of clusters
- More clusters means lower inertia
- What is the best number of clusters?



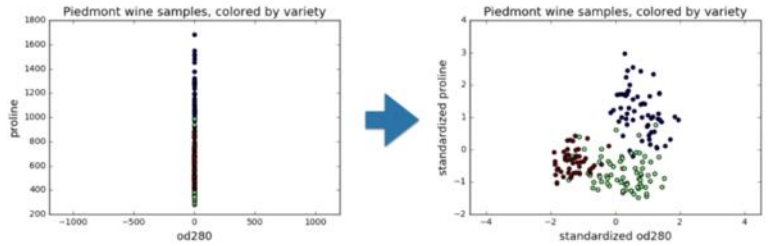
How many clusters to choose?

- A good clustering has tight clusters (so low inertia)
- ... but not too many clusters!
- Choose an "elbow" in the inertia plot
- Where inertia begins to decrease more slowly
- E.g. for iris dataset, 3 is a good choice



StandardScaler

- In kmeans: feature variance = feature influence
- `StandardScaler` transforms each feature to have mean 0 and variance 1
- Features are said to be "standardized"



sklearn StandardScaler

```
In [1]: from sklearn.preprocessing import StandardScaler

In [2]: scaler = StandardScaler()

In [3]: scaler.fit(samples)
Out[3]: StandardScaler(copy=True, with_mean=True, with_std=True)

In [4]: samples_scaled = scaler.transform(samples)
```

Similar methods

- **StandardScaler** and **KMeans** have similar methods
- Use **fit() / transform()** with **StandardScaler**
- Use **fit() / predict()** with **KMeans**

StandardScaler, then KMeans

- Need to perform two steps: **StandardScaler**, then **KMeans**
- Use sklearn pipeline to combine multiple steps
- Data flows from one step into the next

Pipelines combine multiple steps

```
In [1]: from sklearn.preprocessing import StandardScaler

In [2]: from sklearn.cluster import KMeans

In [3]: scaler = StandardScaler()

In [4]: kmeans = KMeans(n_clusters=3)

In [5]: from sklearn.pipeline import make_pipeline

In [6]: pipeline = make_pipeline(scaler, kmeans)

In [7]: pipeline.fit(samples)
Out[7]: Pipeline(steps=...)

In [8]: labels = pipeline.predict(samples)
```

Feature standardization improves clustering

```
In [9]: df = pd.DataFrame({'labels': labels, 'varieties': varieties})

In [10]: ct = pd.crosstab(df['labels'], df['varieties'])

In [11]: print(ct)
varieties  Barbera  Barolo  Grignolino
labels
0                0      59           3
1               48       0           3
2                0       0          65
```

***Without* feature standardization was very bad:**

```
varieties  Barbera  Barolo  Grignolino
labels
0               29      13          20
1                0      46           1
```

sklearn preprocessing steps

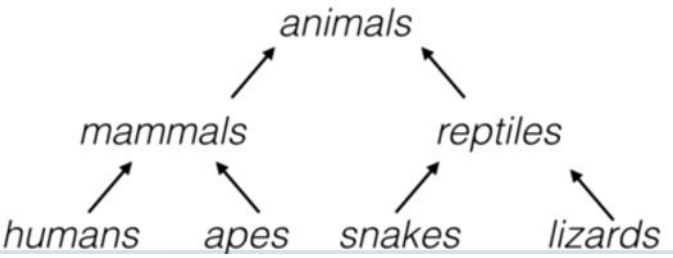
- **StandardScaler** is a "preprocessing" step
- **MaxAbsScaler** and **Normalizer** are other examples

Visualisations communicate insight

- "t-SNE" : Creates a 2D map of a dataset (later)
- "Hierarchical clustering" (this video)

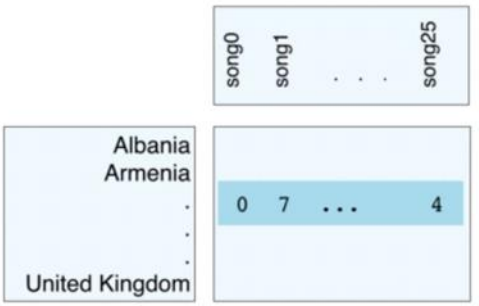
A hierarchy of groups

- Groups of living things can form a hierarchy
- Clusters are contained in one another

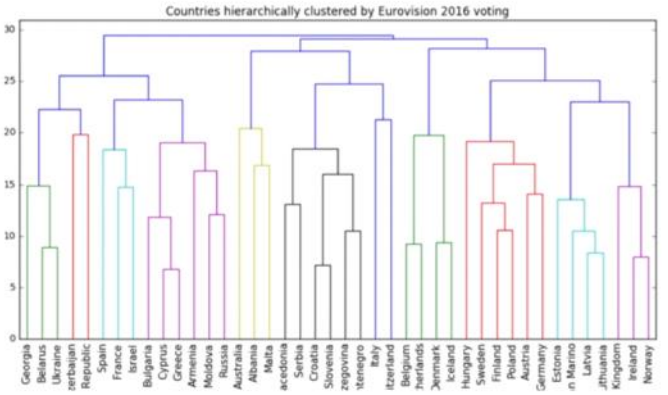


Eurovision scoring dataset

- Countries gave scores to songs performed at the Eurovision 2016
- 2D array of scores
- Rows are countries, columns are songs



Hierarchical clustering of voting countries



Hierarchical clustering

- Every country begins in a separate cluster
- At each step, the two closest clusters are merged
- Continue until all countries in a single cluster
- This is “agglomerative” hierarchical clustering

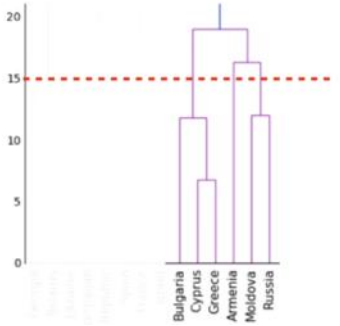
Hierarchical clustering with SciPy

- Given **samples** (the array of scores), and **country_names**

```
In [1]: import matplotlib.pyplot as plt

In [2]: from scipy.cluster.hierarchy import linkage, dendrogram

In [3]: mergings = linkage(samples, method='complete')

In [4]: dendrogram(mergings,
   ...:            labels=country_names,
   ...:            leaf_rotation=90,
   ...:            leaf_font_size=6)

In [5]: plt.show()
```

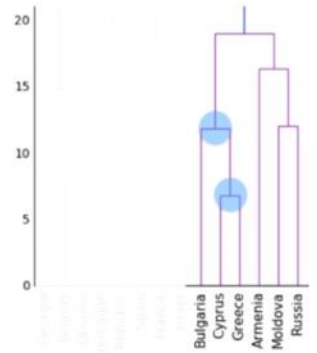
Intermediate clusterings & height on dendrogram

- E.g. at height 15: Bulgaria, Cyprus, Greece are one cluster
- Russia and Moldova are another
- Armenia in a cluster on its own



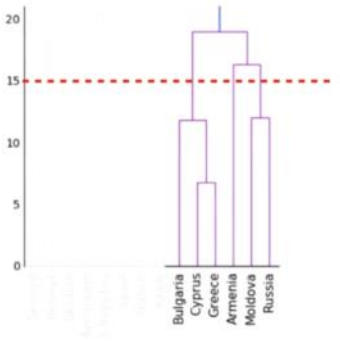
Dendrograms show cluster distances

- Height on dendrogram = distance between merging clusters
- E.g. clusters with only Cyprus and Greece had distance approx. 6
- This new cluster distance approx. 12 from cluster with only Bulgaria



Intermediate clusterings & height on dendrogram

- Height on dendrogram specifies max. distance between merging clusters
- Don't merge clusters further apart than this (e.g. 15)

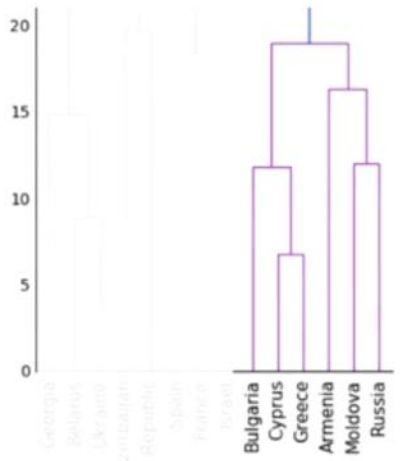


Distance between clusters

- Defined by a "linkage method"
- Specified via method parameter, e.g. **linkage(samples, method="complete")**
- In "complete" linkage: distance between clusters is max. distance between their samples
- Different linkage method, different hierarchical clustering!

Extracting cluster labels

- Use the **fcluster** method
- Returns a NumPy array of cluster labels



Extracting cluster labels using fcluster

```
In [1]: from scipy.cluster.hierarchy import linkage

In [2]: mergings = linkage(samples, method='complete')

In [3]: from scipy.cluster.hierarchy import fcluster

In [4]: labels = fcluster(mergings, 15, criterion='distance')

In [5]: print(labels)
[ 9  8 11 20  2  1 17 14 ... ]
```

Aligning cluster labels with country names

- Given a list of strings **country_names**:

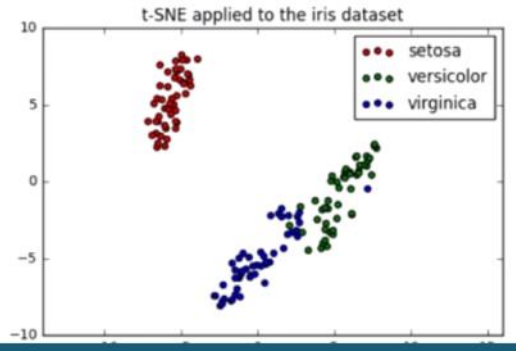
```
In [1]: import pandas as pd

In [2]: pairs = pd.DataFrame({'labels': labels,
   ...:                       'countries': country_names})

In [3]: print(pairs.sort_values('labels'))
              countries  labels
5               Belarus       1
40              Ukraine       1
17              Georgia       1
...
36                Spain       5
8              Bulgaria       6
19               Greece       6
10               Cyprus       6
28              Moldova       7
```

t-SNE for 2-dimensional maps

- t-SNE = “t-distributed stochastic neighbor embedding”
- Maps samples to 2D space (or 3D)
- Map approximately preserves nearness of samples
- Great for inspecting datasets

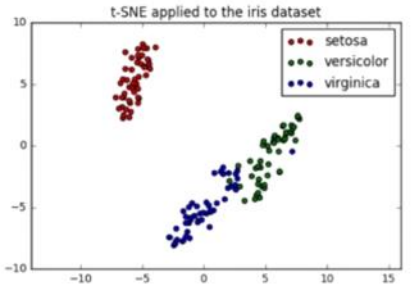
t-SNE on the iris dataset

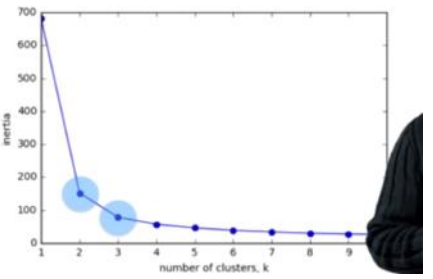
- Iris dataset has 4 measurements, so samples are 4-dimensional
- t-SNE maps samples to 2D space
- t-SNE didn't know that there were different species
- ... yet kept the species mostly separate



Interpreting t-SNE scatter plots

- "versicolor" and "virginica" harder to distinguish from one another
- Consistent with k-means inertia plot: could argue for 2 clusters, or for 3





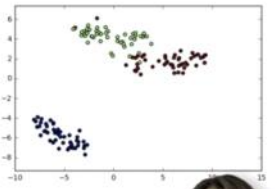
t-SNE in sklearn

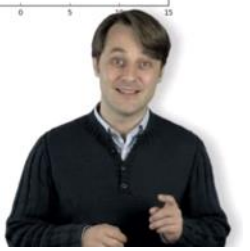
- 2D NumPy array **samples**
- List **species** giving species of labels as number (0, 1, or 2)

```
In [1]: print(samples)
[[ 5.   3.3  1.4  0.2]
 [ 5.   3.5  1.3  0.3]
 [ 4.9  2.4  3.3  1. ]
 [ 6.3  2.8  5.1  1.5]
 ...
 [ 4.9  3.1  1.5  0.1]]

In [2]: print(species)
[0, 0, 1, 2, ..., 0]
```

t-SNE in sklearn

```
In [3]: import matplotlib.pyplot as plt

In [4]: from sklearn.manifold import TSNE

In [5]: model = TSNE(learning_rate=100)

In [5]: transformed = model.fit_transform(samples)

In [6]: xs = transformed[:,0]

In [7]: ys = transformed[:,1]

In [8]: plt.scatter(xs, ys, c=species)

In [9]: plt.show()
```





t-SNE has only fit_transform()

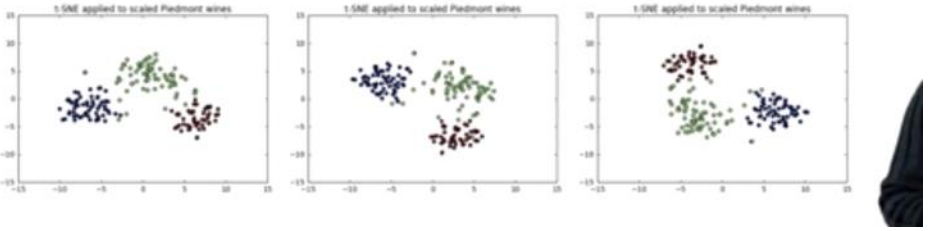
- Has a **fit_transform()** method
- Simultaneously fits the model and transforms the data
- Has no separate **fit()** or **transform()** methods
- Can't extend the map to include new data samples
- Must start over each time!

t-SNE learning rate

- Choose learning rate for the dataset
- Wrong choice: points bunch together
- Try values between 50 and 200

Different every time

- t-SNE features are different every time
- Piedmont wines, 3 runs, 3 different scatter plots!
- ... however: The wine varieties (=colors) have same position relative to one another



Dimension reduction

- More efficient storage and computation
- Remove less-informative "noise" features
- ... which cause problems for prediction tasks, e.g. classification, regression

Principal Component Analysis

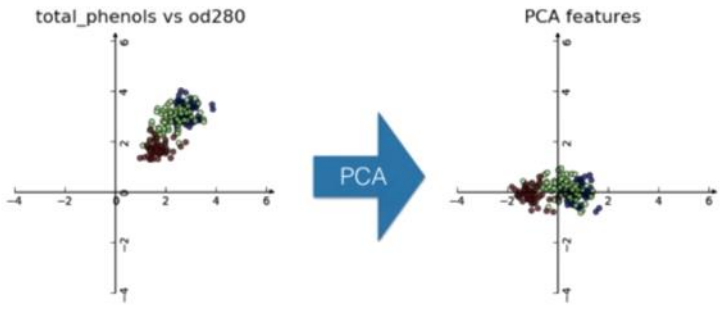
- PCA = "Principal Component Analysis"
- Fundamental dimension reduction technique
- First step "decorrelation" (considered here)
- Second step reduces dimension (considered later)

Principal Component Analysis

- PCA = "Principal Component Analysis"
- Fundamental dimension reduction technique
- First step "decorrelation" (considered here)
- Second step reduces dimension (considered later)

PCA aligns data with axes

- Rotates data samples to be aligned with axes
- Shifts data samples so they have mean 0
- No information is lost



PCA follows the fit/transform pattern

- **PCA** a scikit-learn component like **KMeans** or **StandardScaler**
- **fit()** learns the transformation from given data
- **transform()** applies the learned transformation
- **transform()** can also be applied to new data

Using scikit-learn PCA

- **samples** = array of two wine features (total_phenols & od280)

```
In [1]: print(samples)
[[ 2.8   3.92]
 [ 2.65  3.4 ]
 ...
 [ 2.05  1.6 ]]

In [2]: from sklearn.decomposition import PCA

In [3]: model = PCA()

In [4]: model.fit(samples)
Out[4]: PCA(copy=True, ...)

In [5]: transformed = model.transform(samples)
```

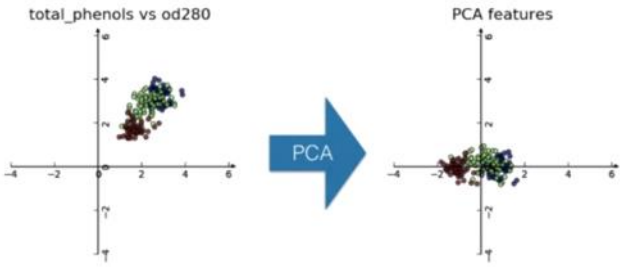
PCA features

- Rows of **transformed** correspond to samples
- Columns of **transformed** are the "PCA features"
- Row gives PCA feature values of corresponding sample

```
In [6]: print(transformed)
[[  1.32771994e+00   4.51396070e-01]
 [  8.32496068e-01   2.33099664e-01]
 ...
 [ -9.33526935e-01  -4.60559297e-01]]
```

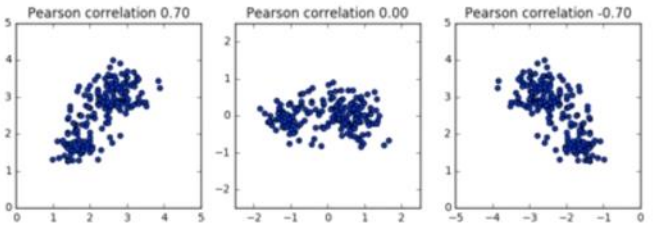
PCA features are not correlated

- Features of dataset are often correlated, e.g. total_phenols and od280
- PCA aligns the data with axes
- Resulting PCA features are not linearly correlated ("decorrelation")



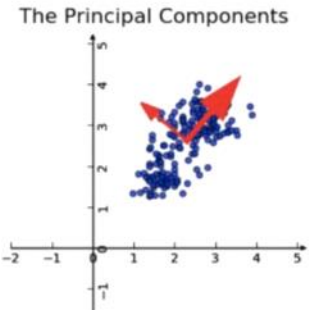
Pearson correlation

- Measures linear correlation of features
- Value between -1 and 1
- Value of 0 means no linear correlation



Principal components

- "Principal components" = directions of variance
- PCA aligns principal components with the axes
- Available as **components_** attribute of PCA object
- Each row defines displacement from mean

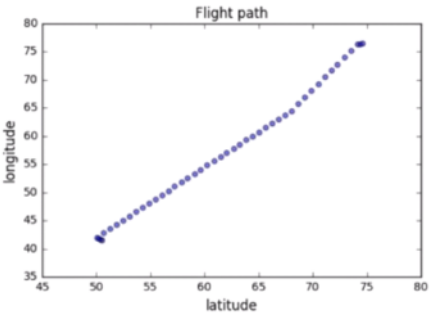



```
In [7]: print(model.components_)
[[ 0.64116665  0.76740167]
 [-0.76740167  0.64116665]]
```

Intrinsic dimension of a flight path

- 2 features: longitude and latitude at points along a flight path
- Dataset *appears* to be 2-dimensional
- But can approximate using one feature: displacement along flight path
- Is intrinsically 1-dimensional

```
latitude  longitude
  50.529     41.513
  50.360     41.672
  50.196     41.835
  50.035     42.015
  50.678     42.796
  51.281     43.526
...
```



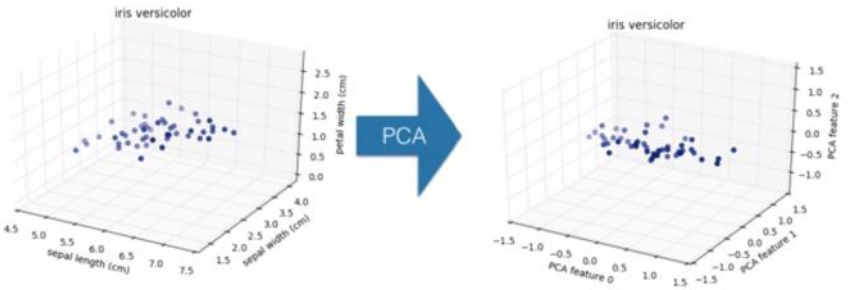
Intrinsic dimension

- Intrinsic dimension = number of features needed to approximate the dataset
- Essential idea behind dimension reduction
- What is the most compact representation of the samples?
- Can be detected with PCA

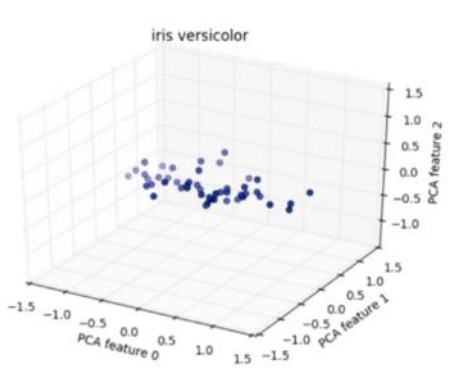
PCA identifies intrinsic dimension

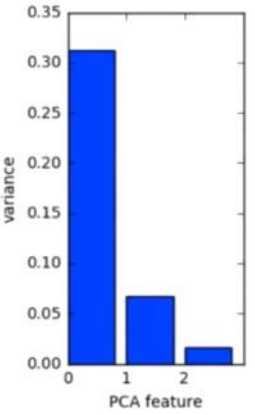
- Scatter plots work only if samples have 2 or 3 features
- PCA identifies intrinsic dimension when samples have *any number* of features
- Intrinsic dimension = number of PCA features with significant variance

PCA of the versicolor samples



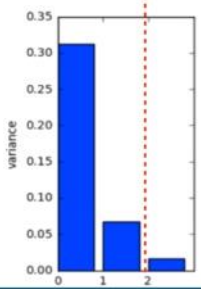
PCA features are ordered by variance descending





Variance and intrinsic dimension

- Intrinsic dimension is number of PCA features with significant variance
- In our example: the first two PCA features
- So intrinsic dimension is 2



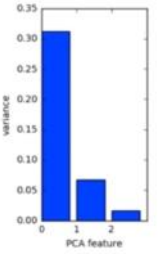
Plotting the variances of PCA features

- **samples** = array of versicolor samples

```
In [1]: import matplotlib.pyplot as plt

In [2]: from sklearn.decomposition import PCA

In [3]: pca = PCA()

In [4]: pca.fit(samples)
Out[4]: PCA(copy=True, ... )

In [5]: features = range(pca.n_components_)
```

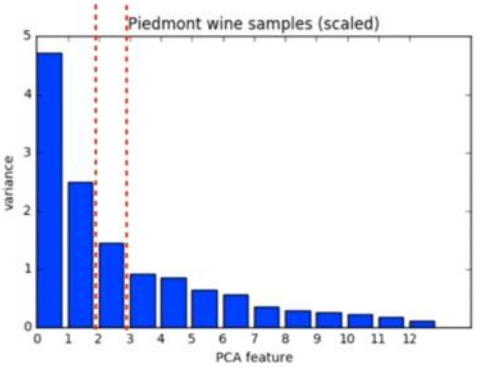
Plotting the variances of PCA features

```
In [6]: plt.bar(features, pca.explained_variance_)

In [7]: plt.xticks(features)

In [8]: plt.ylabel('variance')

In [9]: plt.xlabel('PCA feature')

In [10]: plt.show()
```



Intrinsic dimension can be ambiguous

- Intrinsic dimension is an idealization
- ... there is not always one correct answer!
- Piedmont wines: could argue for 2, or for 3, or more

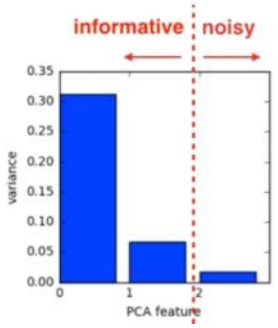


Dimension reduction

- Represents same data, using less features
- Important part of machine-learning pipelines
- Can be performed using PCA

Dimension reduction with PCA

- PCA features are in decreasing order of variance
- Assumes the low variance features are "noise"
- ... and high variance features are informative



Dimension reduction with PCA

- Specify how many features to keep
- E.g. **PCA(n_components=2)**
- Keeps the first 2 PCA features
- Intrinsic dimension is a good choice

Dimension reduction of iris dataset

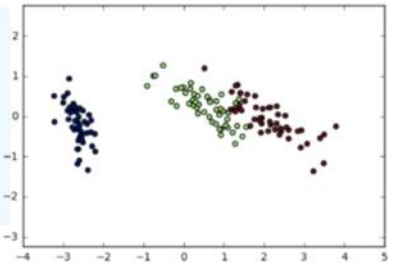
- **samples** = array of iris measurements (4 features)
- **species** = list of iris species numbers

```
In [1]: from sklearn.decomposition import PCA

In [2]: pca = PCA(n_components=2)

In [3]: pca.fit(samples)
Out[3]: PCA(copy=True, ... )

In [4]: transformed = pca.transform(samples)

In [5]: print(transformed.shape)
(150, 2)
```

Iris dataset in 2 dimensions

- PCA has reduced the dimension to 2
- Retained the 2 PCA features with highest variance
- Important information preserved: species remain distinct

```
In [6]: import matplotlib.pyplot as plt

In [7]: xs = transformed[:,0]
   ...: ys = transformed[:,1]

In [8]: plt.scatter(xs, ys, c=species)
   ...: plt.show()
```

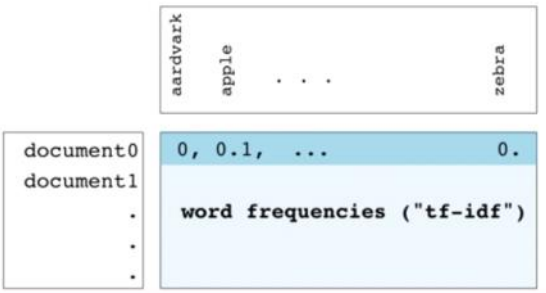


Dimension reduction with PCA

- Discards low variance PCA features
- Assumes the high variance features are informative
- Assumption typically holds in practice (e.g. for iris)

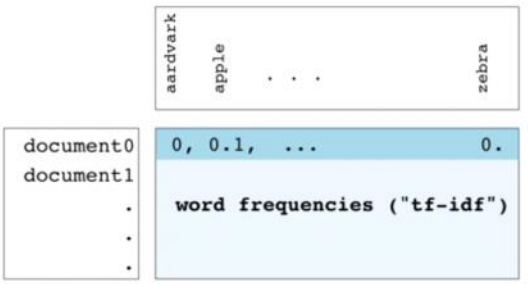
Word frequency arrays

- Rows represent documents, columns represent words
- Entries measure presence of each word in each document
- ... measure using "tf-idf" (more later)



Sparse arrays and csr_matrix

- Array is "sparse": most entries are zero
- Can use **scipy.sparse.csr_matrix** instead of NumPy array
- **csr_matrix** remembers only the non-zero entries (saves space!)



TruncatedSVD and csr_matrix

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

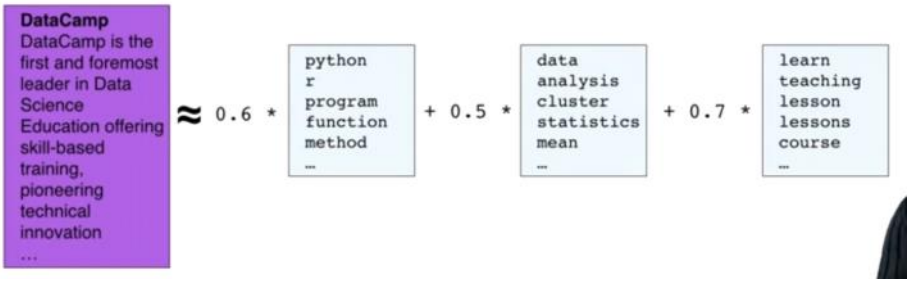
```
In [1]: from sklearn.decomposition import TruncatedSVD

In [2]: model = TruncatedSVD(n_components=3)

In [3]: model.fit(documents)  # documents is csr_matrix
Out[3]: TruncatedSVD(algorithm='randomized', ... )

In [4]: transformed = model.transform(documents)
```

Non-negative matrix factorization

- NMF = "non-negative matrix factorization"
- Dimension reduction technique
- NMF models are *interpretable* (unlike PCA)
- Easy to interpret means easy to explain!
- However, all sample features must be non-negative (>= 0)

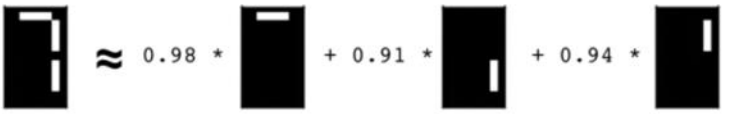
Interpretable parts

- NMF expresses documents as combinations of topics (or "themes")



Interpretable parts

- NMF expresses images as combinations of patterns



Using scikit-learn NMF

- Follows **fit()** / **transform()** pattern
- Must specify number of components e.g. **NMF(n_components=2)**
- Works with NumPy arrays and with **csr_matrix**

Example word-frequency array

- Word frequency array, 4 words, many documents
- Measure presence of words in each document using "tf-idf"
- "tf" = frequency of word in document
- "idf" reduces influence of frequent words

	course	datacamp	potato	the
document0	0.2,	0.3,	0.0,	0.1
document1	0.0,	0.0,	0.4,	0.1
...		...		

Example usage of NMF

- **samples** is the word-frequency array

```
In [1]: from sklearn.decomposition import NMF

In [2]: model = NMF(n_components=2)

In [3]: model.fit(samples)
Out[3]: NMF(alpha=0.0, ... )

In [4]: nmf_features = model.transform(samples)
```

NMF components

- NMF has components
- ... just like PCA has principal components
- Dimension of components = dimension of samples
- Entries are non-negative

```
In [5]: print(model.components_)
[[ 0.01  0.    2.13  0.54]
 [ 0.99  1.47  0.    0.5 ]]
```

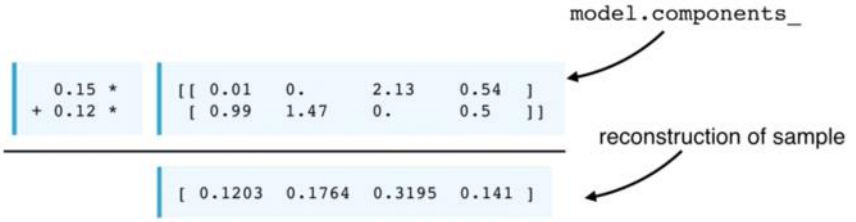
NMF features

- NMF feature values are non-negative
- Can be used to reconstruct the samples
- ... combine feature values with components

```
In [6]: print(nmf_features)
[[ 0.    0.2 ]
 [ 0.19  0.  ]
  ...
 [ 0.15  0.12]]
```

Reconstruction of a sample

```
In [7]: print(samples[i,:])
[ 0.12  0.18  0.32  0.14]

In [8]: print(nmf_features[i,:])
[ 0.15  0.12]
```



Sample reconstruction

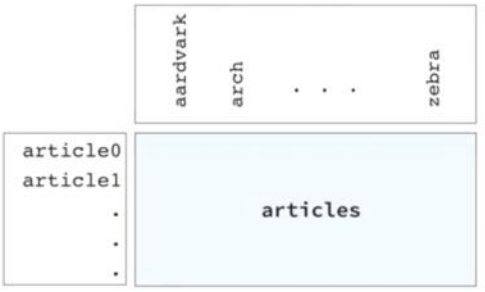
- Multiply components by feature values, and add up
- Can also be expressed as a product of matrices
- This is the "**M**atrix **F**actorization" in "NMF"

NMF fits to non-negative data, only

- Word frequencies in each document
- Images encoded as arrays
- Audio spectrograms
- Purchase histories on e-commerce sites
- ... and many more!

Example: NMF learns interpretable parts

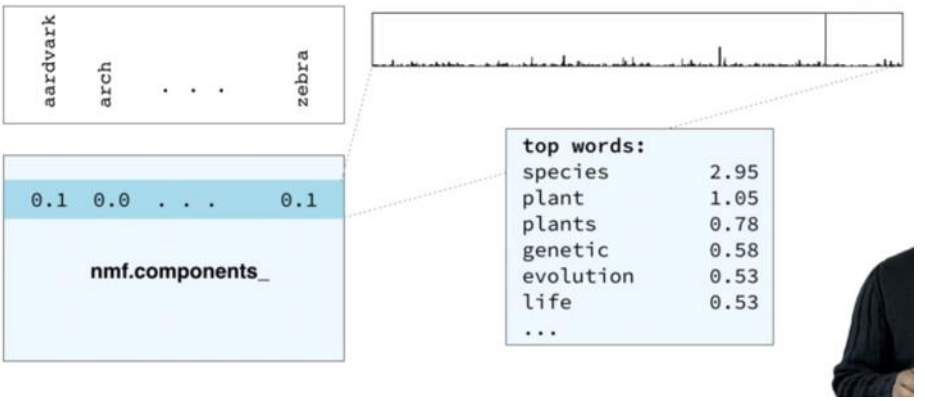
- Word-frequency array **articles** (tf-idf)
- 20,000 scientific articles (rows)
- 800 words (columns)



Applying NMF to the articles

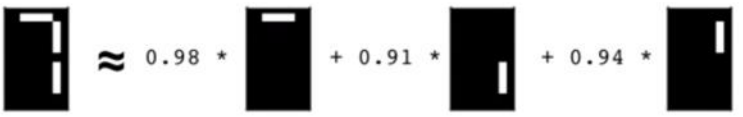
```
In [1]: print(articles.shape)
(20000, 800)

In [2]: from sklearn.decomposition import NMF

In [3]: nmf = NMF(n_components=10)

In [4]: nmf.fit(articles)
Out[4]: NMF(alpha=0.0, ... )

In [5]: print(nmf.components_.shape)
(10, 800)
```

NMF components are topics



NMF components

- For documents:
  - NMF components represent topics
  - NMF features combine topics into documents
- For images, NMF components are parts of images



Grayscale images

- "Grayscale" image = no colors, only shades of gray
- Measure pixel brightness
- Represent with value between 0 and 1 (0 is black)
- Convert to 2D array

```
[[ 0.   1.   0.5]
 [ 1.   0.   1. ]]
```

Grayscale image example

- An 8x8 grayscale image of the moon, written as an array

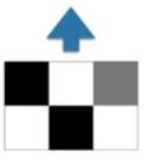
```
[[ 0.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.7  0.8  0.   0.   0. ]
 [ 0.   0.   0.8  0.8  0.9  1.   0.   0. ]
 [ 0.   0.7  0.9  0.9  1.   1.   1.   0. ]
 [ 0.   0.8  0.9  1.   1.   1.   1.   0. ]
 [ 0.   0.   0.9  1.   1.   1.   0.   0. ]
 [ 0.   0.   0.   0.9  1.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0. ]]
```

Grayscale images as flat arrays

- Enumerate the entries
- Row-by-row
- From left to right

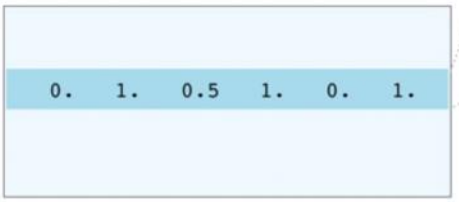
```
[[ 0.   1.   0.5]
 [ 1.   0.   1. ]]
```

```
[ 0.   1.   0.5   1.   0.   1. ]
```



Encoding a collection of images

- Collection of images of the same size
- Encode as 2D array
- Each row corresponds to an image
- Each column corresponds to a pixel
- ... can apply NMF!



Visualizing samples

```
In [1]: print(sample)
[ 0.   1.   0.5  1.   0.   1. ]

In [2]: bitmap = sample.reshape((2, 3))

In [3]: print(bitmap)
[[ 0.   1.   0.5]
 [ 1.   0.   1. ]]

In [4]: from matplotlib import pyplot as plt

In [5]: plt.imshow(bitmap, cmap='gray', interpolation='nearest')

In [6]: plt.show()
```

Finding similar articles

- Engineer at a large online newspaper
- Task: recommend articles similar to article being read by customer
- Similar articles should have similar topics

Strategy

- Apply NMF to the word-frequency array
- NMF feature values describe the topics
- ... so similar documents have similar NMF feature values
- Compare NMF feature values?

Apply NMF to the word-frequency array

- **articles** is a word frequency array

```
In [1]: from sklearn.decomposition import NMF

In [2]: nmf = NMF(n_components=6)

In [3]: nmf_features = nmf.fit_transform(articles)
```

Versions of articles

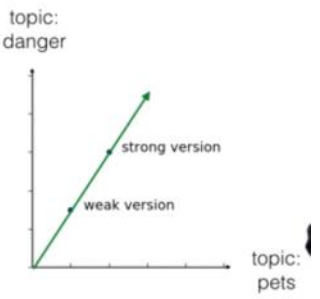
- Different versions of the same document have same topic *proportions*
- ... exact feature values may be different!
- E.g. because one version uses many meaningless words
- But all versions lie on the same *line* through the origin

strong version

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

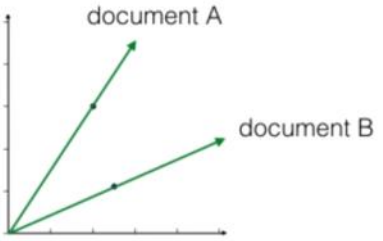
weak version

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



Cosine similarity

- Uses the angle between the lines
- Higher values means more similar
- Maximum value is 1, when angle is 0°



Calculating the cosine similarities

```
In [4]: from sklearn.preprocessing import normalize

In [5]: norm_features = normalize(nmf_features)

In [6]: current_article = norm_features[23,:]  # if has index 23

In [7]: similarities = norm_features.dot(current_article)

In [8]: print(similarities)
[ 0.7150569   0.26349967  0.40210445 ...,  0.70462768  0.20323616
  0.05047817]
```

DataFrames and labels

- Label similarities with the article titles, using a DataFrame
- Titles given as a list: **titles**

```
In [9]: import pandas as pd

In [10]: norm_features = normalize(nmf_features)

In [11]: df = pd.DataFrame(norm_features, index=titles)

In [12]: current_article = df.loc['Dog bites man']

In [13]: similarities = df.dot(current_article)
```

DataFrames and labels

```
In [14]: print(similarities.nlargest())
Dog bites man                          1.000000
Hound mauls cat                        0.979946
Pets go wild!                          0.979708
Dachshunds are dangerous               0.949641
Our streets are no longer safe         0.900474
dtype: float64
```

Linear classifiers in python

Thursday, 19 March 2020 3:06 PM

Assumed knowledge

In this course we'll assume you have some prior exposure to:

- Python, at the level of *Intermediate Python for Data Science*
- scikit-learn, at the level of *Supervised Learning with scikit-learn*
- supervised learning, at the level of *Supervised Learning with scikit-learn*

Fitting and predicting

```
import sklearn.datasets

newsgroups = sklearn.datasets.fetch_20newsgroups_vectorized()

X, y = newsgroups.data, newsgroups.target
```

```
X.shape
```

```
(11314, 130107)
```

```
y.shape
```

```
(11314,)
```

In this case features are derived from words appearing in each news article and y values are The article topics which is what we are trying to predict

Fitting and predicting (cont.)

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=1)
```

```
knn.fit(X,y)
```

```
y_pred = knn.predict(X)
```

Model evaluation

```
knn.score(X,y)
```

```
0.99991
```

```
from sklearn.model_selection import train_test_split

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

```
knn.fit(X_train, y_train)

knn.score(X_test, y_test)
```

```
0.66242
```

Using LogisticRegression

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

```
lr = LogisticRegression()
lr.fit(X_train, y_train)
lr.predict(X_test)
lr.score(X_test, y_test)
```

LogisticRegression example

```
import sklearn.datasets
wine = sklearn.datasets.load_wine()
```

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]])
```

Using LinearSVC

`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
```

Using SVC

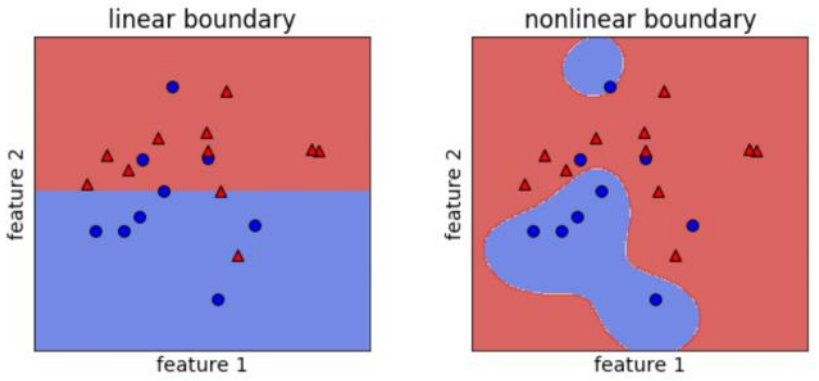
```
import sklearn.datasets
wine = sklearn.datasets.load_wine()
from sklearn.svm import SVC
svm = SVC() # default hyperparameters
svm.fit(wine.data, wine.target);
svm.score(wine.data, wine.target)
```

1.

Model complexity review:

- **Underfitting**: model is too simple, low training accuracy
- **Overfitting**: model is too complex, low test accuracy

Linear decision boundaries

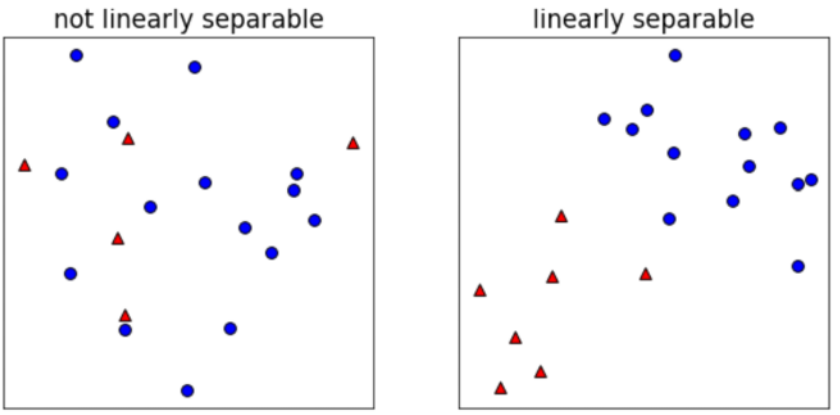


Definitions

Vocabulary:

- **classification**: learning to predict categories
- **decision boundary**: the surface separating different predicted classes
- **linear classifier**: a classifier that learns linear decision boundaries
  - e.g., logistic regression, linear SVM
- **linearly separable**: a data set can be perfectly explained by a linear classifier

Linearly separable data



Dot Products

```
x = np.arange(3)
x
```

```
array([0, 1, 2])
```

```
y = np.arange(3,6)
y
```

```
array([3, 4, 5])
```

```
x*y
```

```
array([0, 4, 10])
```

```
np.sum(x*y)
```

```
14
```

```
x@y
```

```
14
```

- `x@y` is called the dot product of `x` and `y`, and is written $x \cdot y$.

Linear classifier prediction

- $\text{raw model output} = \text{coefficients} \cdot \text{features} + \text{intercept}$
- Linear classifier prediction: compute raw model output, check the **sign**
  - if positive, predict one class
  - if negative, predict the other class
- This is the same for logistic regression and linear SVM
  - `fit` is different but `predict` is the same

How LogisticRegression makes predictions

$\text{raw model output} = \text{coefficients} \cdot \text{features} + \text{intercept}$

```
lr = LogisticRegression()

lr.fit(X,y)

lr.predict(X)[10]
```

```
0
```

```
lr.predict(X)[20]
```

```
1
```

How LogisticRegression makes predictions (cont.)

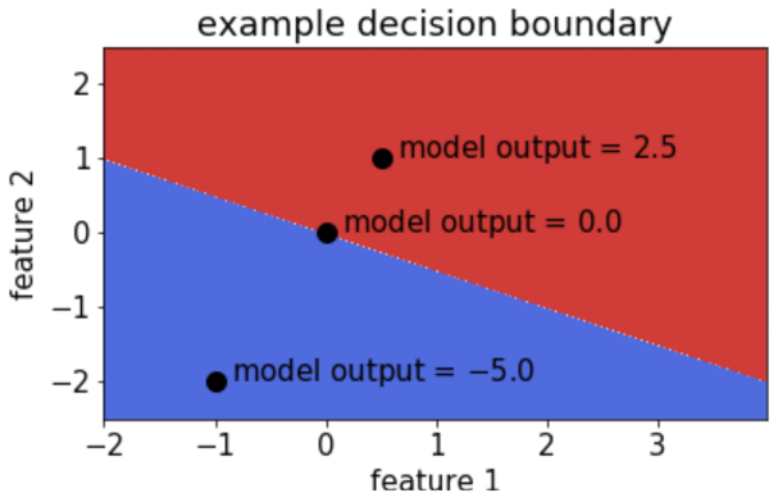
```
lr.coef_ @ X[10] + lr.intercept_ # raw model output
```

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

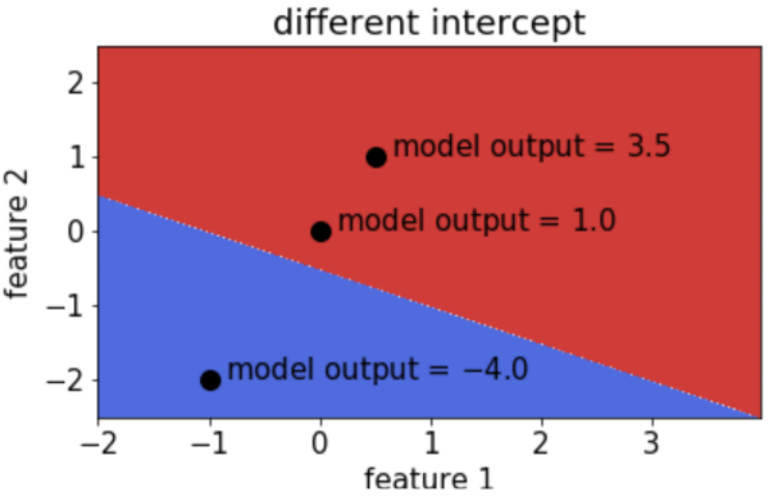
```
lr.coef_ @ X[20] + lr.intercept_ # raw model output
```

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

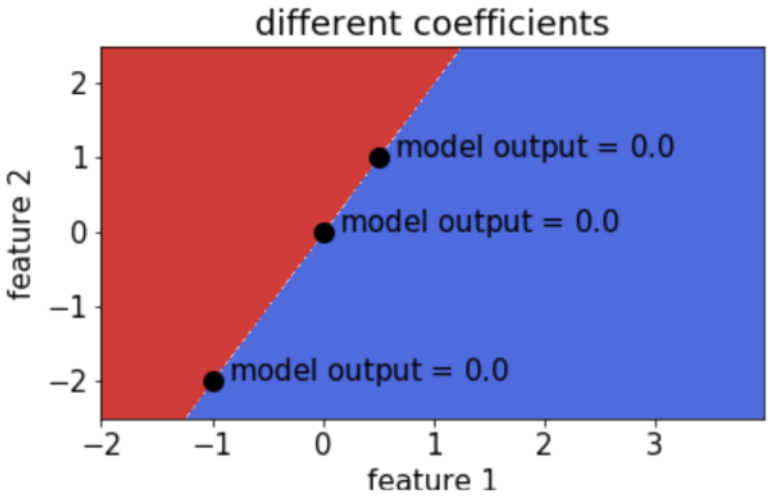
The raw model output



The raw model output



The raw model output



Least squares: the squared loss

- scikit-learn's `LinearRegression` minimizes a loss:

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

- Minimization is with respect to coefficients or parameters of the model.
- Note that in scikit-learn `model.score()` isn't necessarily the loss function.

Classification errors: the 0-1 loss

- Squared loss not appropriate for classification problems (more on this later).
- A natural loss for classification problem is the number of errors.
- This is the **0-1 loss**: it's 0 for a correct prediction and 1 for an incorrect prediction.
- But this loss is hard to minimize!

Minimizing a loss

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

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

Minimizing a loss

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

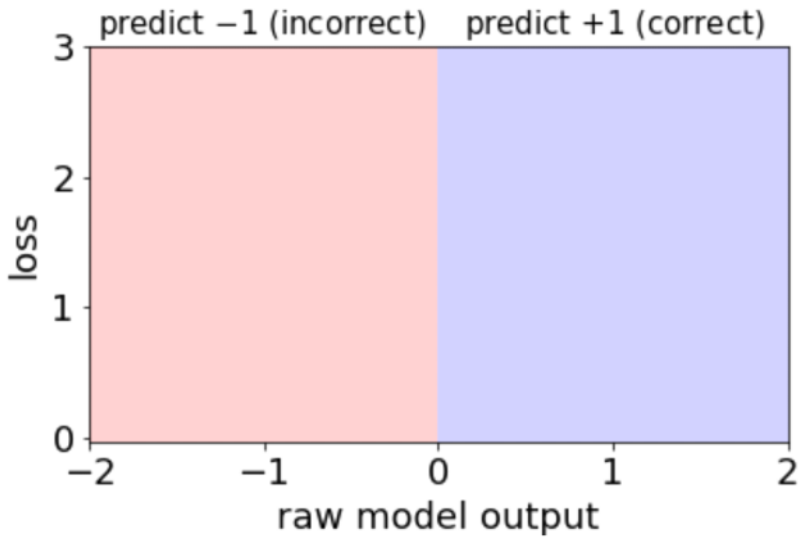
```
minimize(np.square, 0).x
```

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

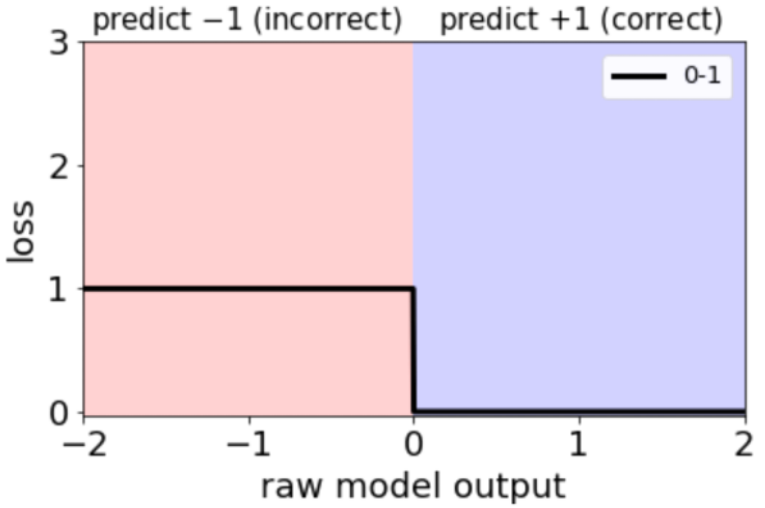
```
minimize(np.square, 2).x
```

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

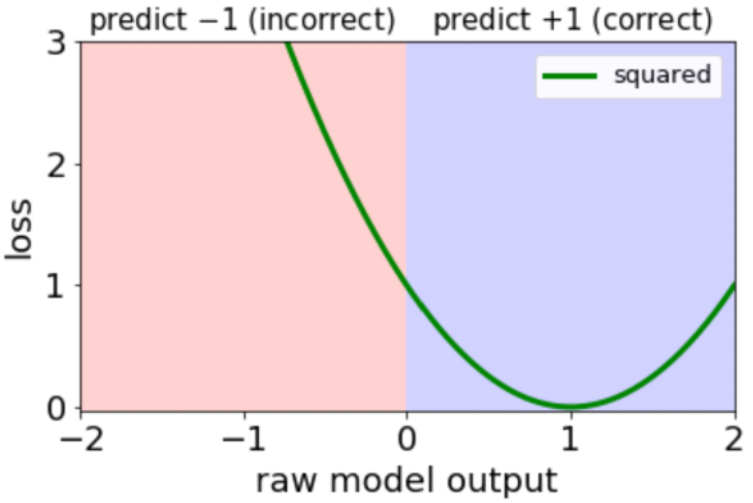
The raw model output



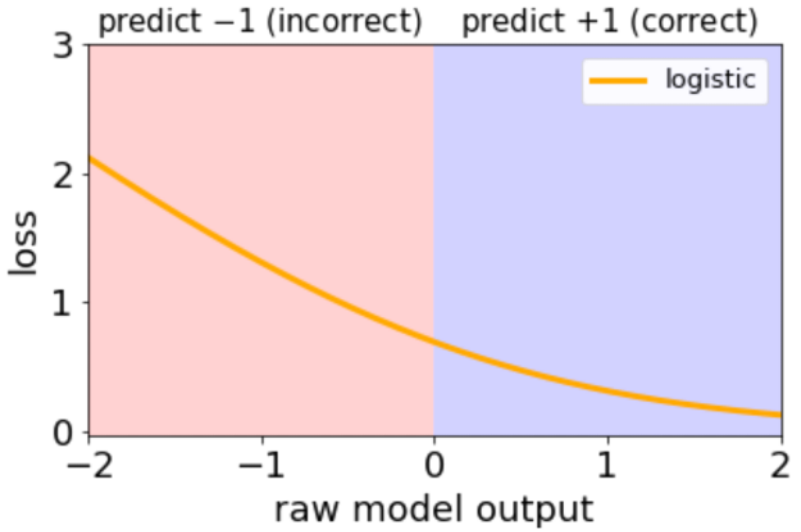
0-1 loss diagram



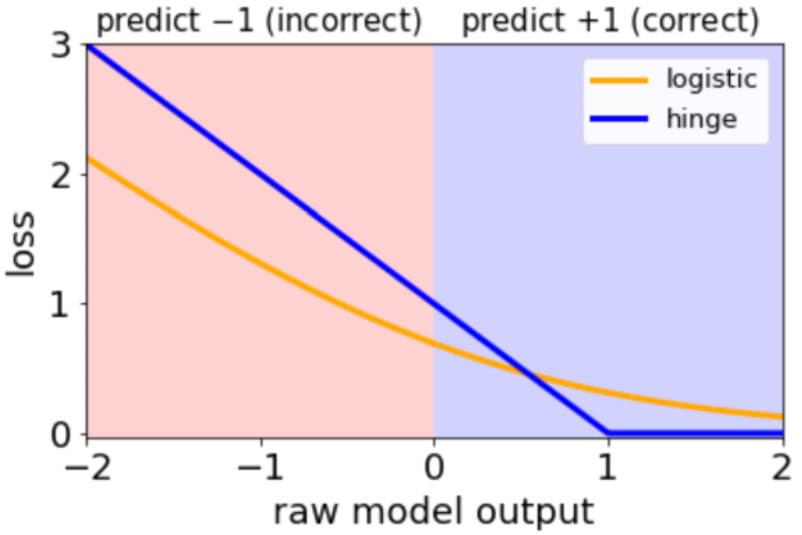
Linear regression loss diagram



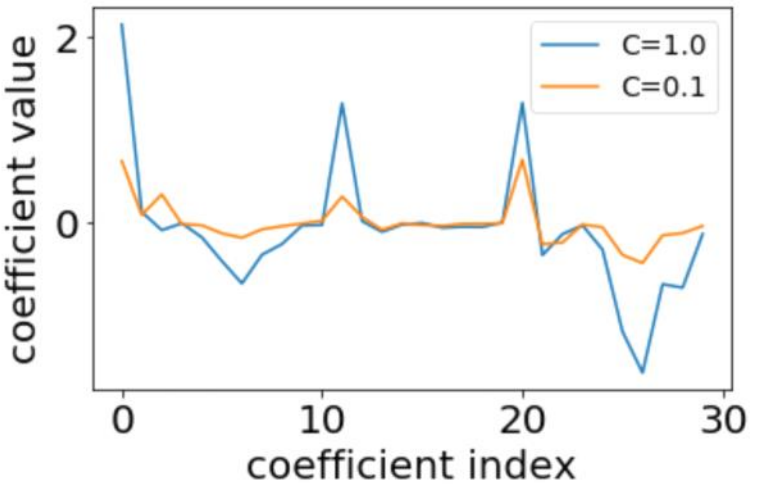
Logistic loss diagram



Hinge loss diagram



Regularized logistic regression



How does regularization affect training accuracy?

```
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
```

regularized loss = original loss + large coefficient penalty

- more regularization: lower training accuracy

How does regularization affect test accuracy?

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

```
0.86
```

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

```
0.88
```

regularized loss = original loss + large coefficient penalty

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

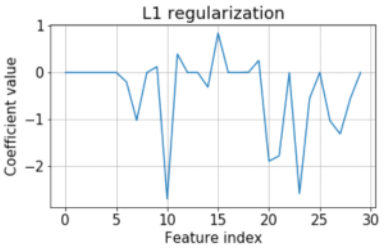
L1 vs. L2 regularization

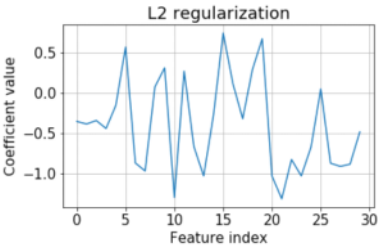
- Lasso = linear regression with L1 regularization
- Ridge = linear regression with L2 regularization
- For other models like logistic regression we just say L1, L2, etc.

```
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())
```

L2 vs. L1 regularization

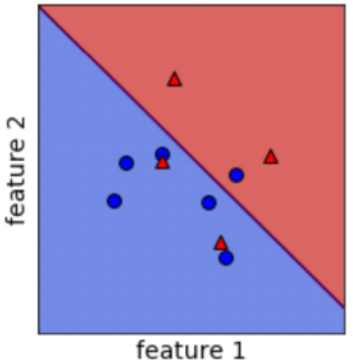




Logistic regression probabilities

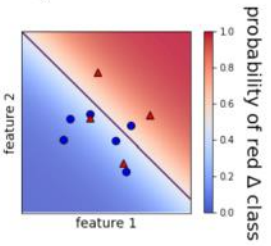
Without regularization ($C = 10^8$):

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



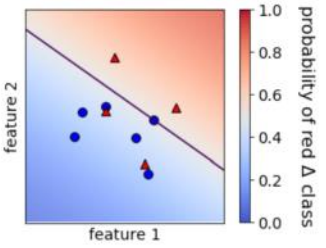
Logistic regression probabilities

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



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

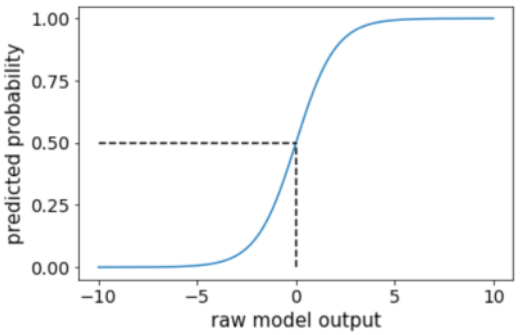
With regularization $(C = 1)$:



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

How are these probabilities computed?

- logistic regression predictions: sign of raw model output
- logistic regression probabilities: "squashed" raw model output



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
```

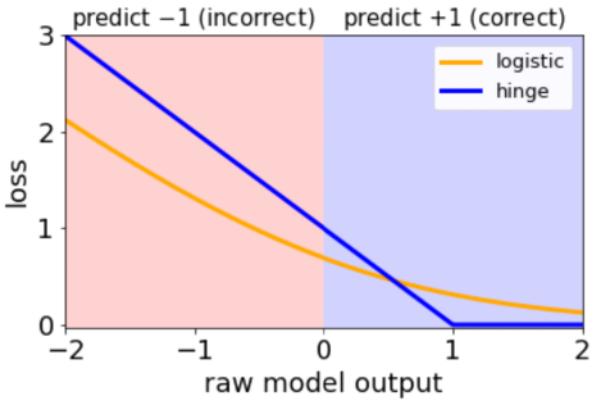
```
(3,13)
```

```
lr_mn.intercept_.shape
```

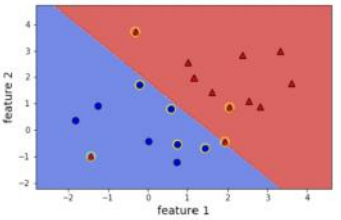
```
(3,)
```

What is an SVM?

- Linear classifiers (so far)
- Trained using the hinge loss and L2 regularization

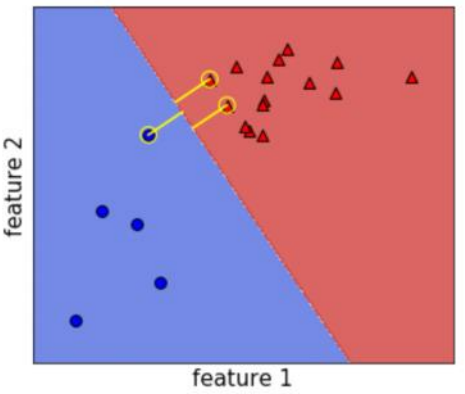


- Support vector: a training example **not** in the flat part of the loss diagram
- Support vector: an example that is incorrectly classified **or** close to the boundary
- If an example is not a support vector, removing it has no effect on the model
- Having a small number of support vectors makes kernel SVMs really fast

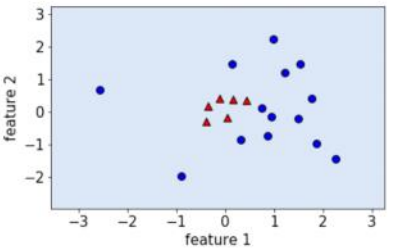


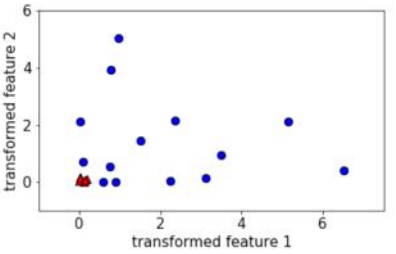
Max-margin viewpoint

- The SVM maximizes the "margin" for linearly separable datasets
- Margin: distance from the boundary to the closest points



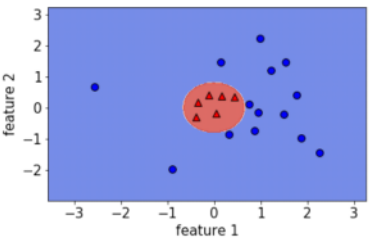
Transforming your features

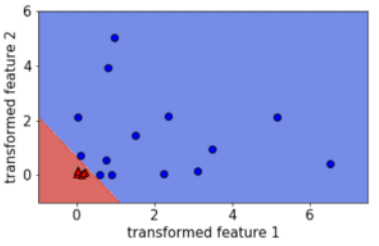




$$\text{transformed feature} = (\text{original feature})^2$$

Transforming your features

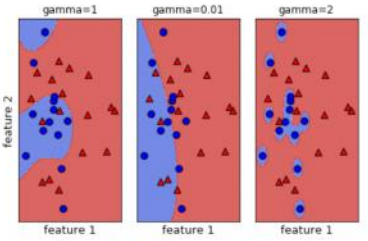




$$\text{transformed feature} = (\text{original feature})^2$$

Kernel SVMs

```
from sklearn.svm import SVC

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



- larger `gamma` leads to more complex boundaries

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

Use in scikit-learn

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`

Use in scikit-learn (cont.)

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`