COMP 364: Computer Tools for Life Sciences

Intro to machine learning with scikit-learn (part two)

Christopher J.F. Cameron and Carlos G. Oliver

Key course information

Quiz #8

- available now
- due Monday, November 20th at 11:59:59 pm
- covers topics from the last two weeks

Course evaluations

- available now at the following link:
  - `https://horizon.mcgill.ca/pban1/twbkwbis.P_WWWLogin?ret_code=f`

Under/overfitting

Under/overfitting are important concepts in machine learning

- to ensure everyone understands
- we're going to do a more elaborate example

Let's start by generating some data

- follows a cosine with some added random noise

```
1 import numpy as np
2
3 n_samples = 100
4 X = np.sort(np.random.rand(n_samples))
5 y = np.cos(1.5 * np.pi * X)
6     + np.random.randn(n_samples) * 0.1
```

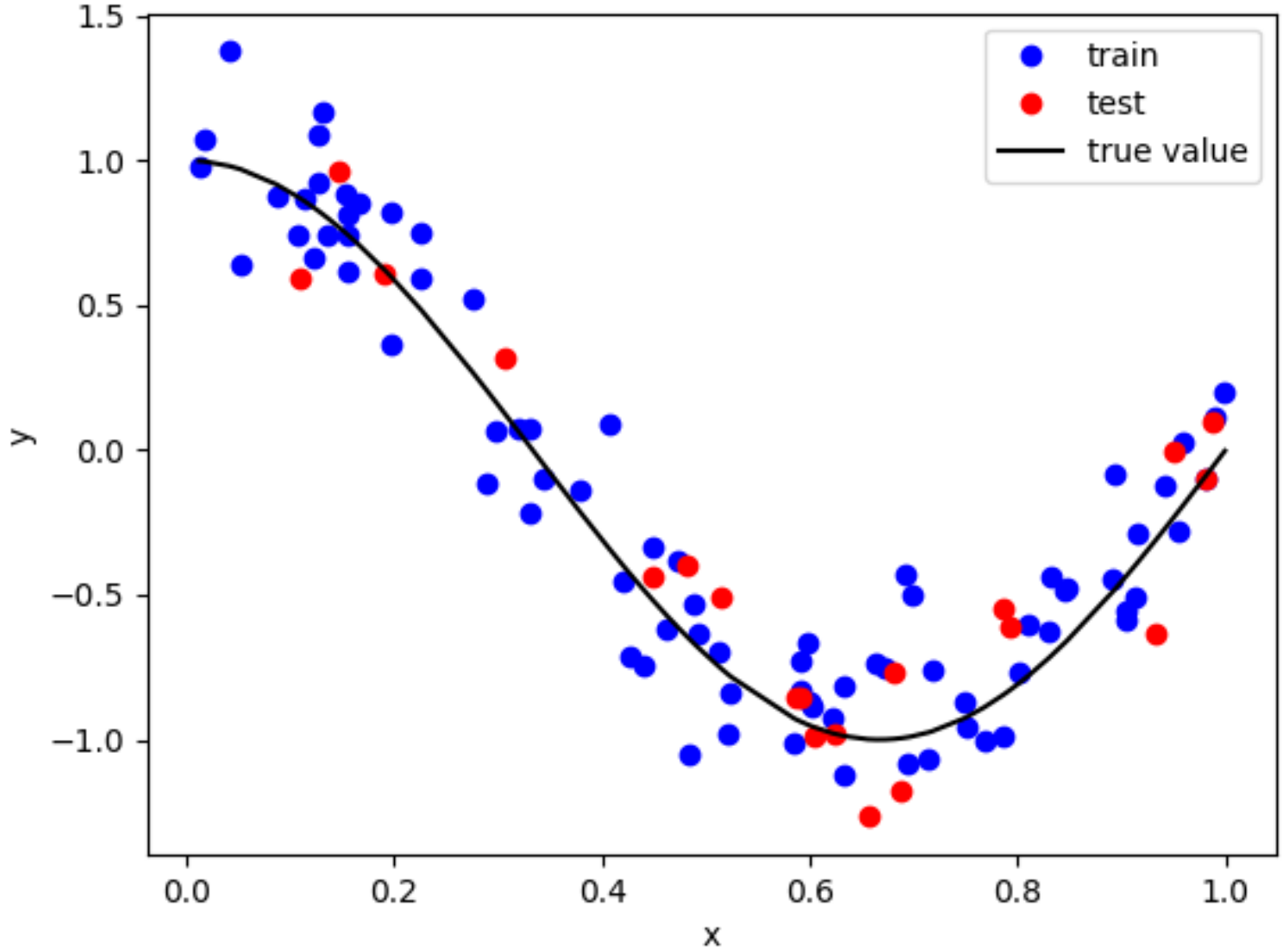
Overfitting example

Treat the data as if it is a **machine learning (ML)** problem

- split into **training** and **testing** datasets

```
1  import numpy as np
2  from sklearn import model_selection
3  # split data into training and test datasets
4  X_train,X_test = model_selection.train_test_split(X,
5                      test_size = 0.2, shuffle = True)
6  # sort data for plotting
7  X_train,X_test = np.sort(X_train),np.sort(X_test)
8  y_train = np.cos(1.5 * np.pi * X_train)
9      + np.random.randn(80) * 0.2
10 y_test = np.cos(1.5 * np.pi * X_test)
11     + np.random.randn(20) * 0.2
```

Let's also plot the data so that we have a better understanding


train
test
true value
1.5
1.0
0.5
0.0
−0.5
−1.0
y
0.0
0.2
0.4
0.6
0.8
1.0
x

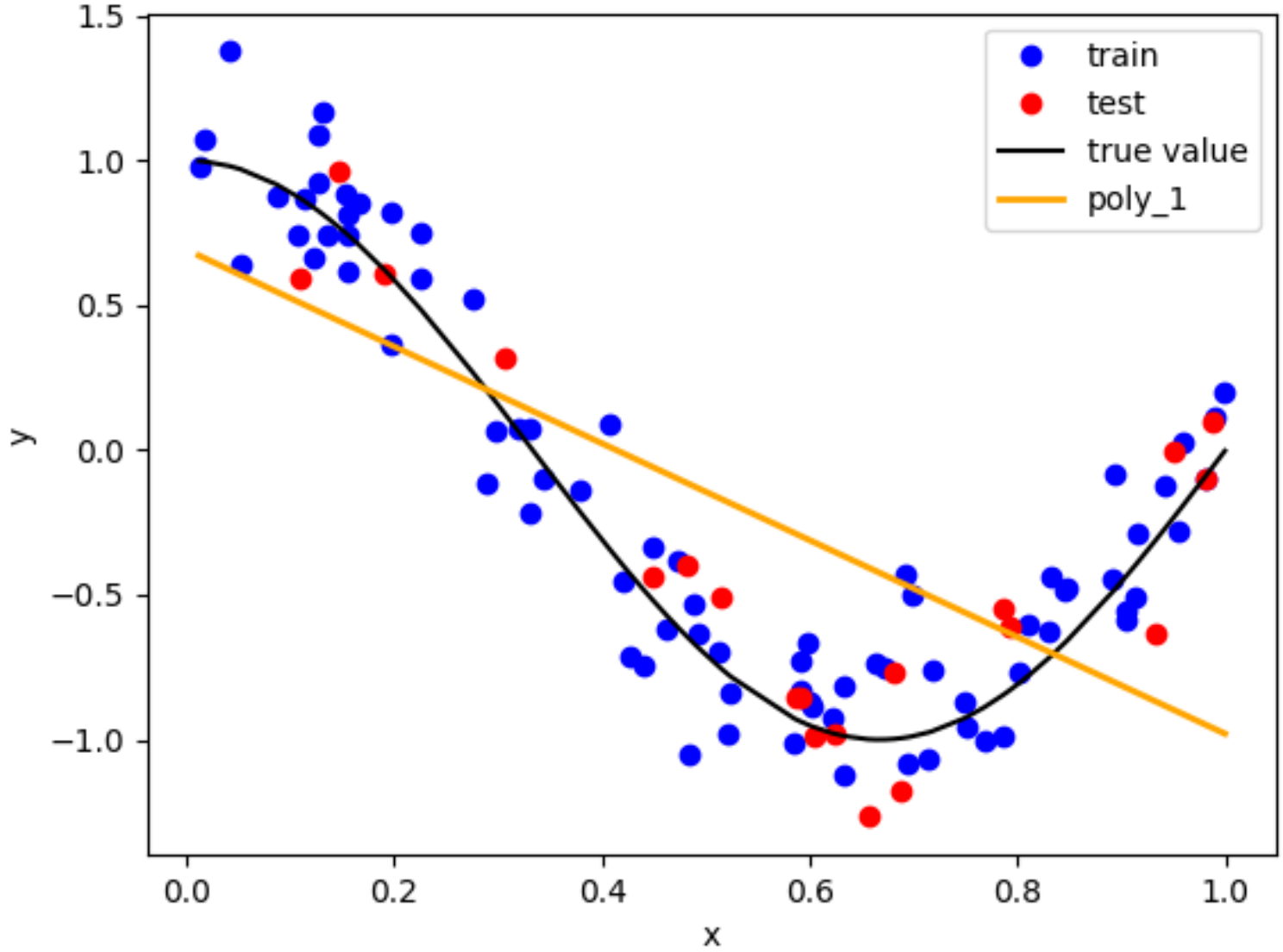
Overfitting example #2

Now let's try to 'learn' the training data distribution

- by drawing a line that best fits the training data
- or by performing linear regression

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# transform data into matrices for regression
reg_X_train = X_train[:,np.newaxis]
pipeline = Pipeline([("polynomial_features",
PolynomialFeatures(1, include_bias=False))
 ,("linear_regression", LinearRegression())])
 pipeline.fit(reg_X_train, y_train)
preds_train = pipeline.predict(reg_X_train)
```


train
test
true value
poly_1
1.5
1.0
0.5
0.0
−0.5
−1.0
y
0.0
0.2
0.4
0.6
0.8
1.0
x

Accuracy measures

How can we determine the accuracy of a model's predictions?

- linear regression is just like ML
- we have a set of examples (X) with known labels (y)
- slope and coefficient intercepts are calculated to fit a $func()$
- where $y = func(X) = mX + b$

The scikit-learn module has many accuracy metrics to choose from:

- `http://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics`

For today, we'll stick to the **mean squared error (MSE)**

Mean squared error (MSE)

To calculate:

$$MSE = \frac{1}{N} \sum_{i=1}^{N} \left(\hat{Y}_i - Y_i \right)^2$$

where:

- N is the number of examples
- Y is the known label for input X_i
- $\hat{Y}_i$ is a model's prediction for input X_i

A model is considered to be more accurate as MSE approaches zero

Question: why do we square $\hat{Y}_i - Y_i$?

Calculating MSE

With scikit-learn, MSE can be calculated using the function below:

```
from sklearn.metrics import mean_squared_error

pipeline.fit(reg_X_train, y_train)
preds_train = pipeline.predict(reg_X_train)
train_err = mean_squared_error(y_train,preds_train)
reg_X_test = X_test[:,np.newaxis]
test_err = mean_squared_error(y_test,
  pipeline.predict(reg_X_test))
print(degree,train_err,test_err)
#prints: 1 0.237980438732 0.310466053348
```

Calculating MSE #2

Linear regression

- training MSE: 0.237980438732
- testing MSE: 0.310466053348

How do we interpret these numbers?

- well, like most things in science, with relativity
- both training and testing datasets have similar MSE
- from the plot, the line does not fit either dataset that well
- we would conclude that underfitting has occurred
  - the model is unable to extract significant data from examples to learn the training distribution

More complex regressions

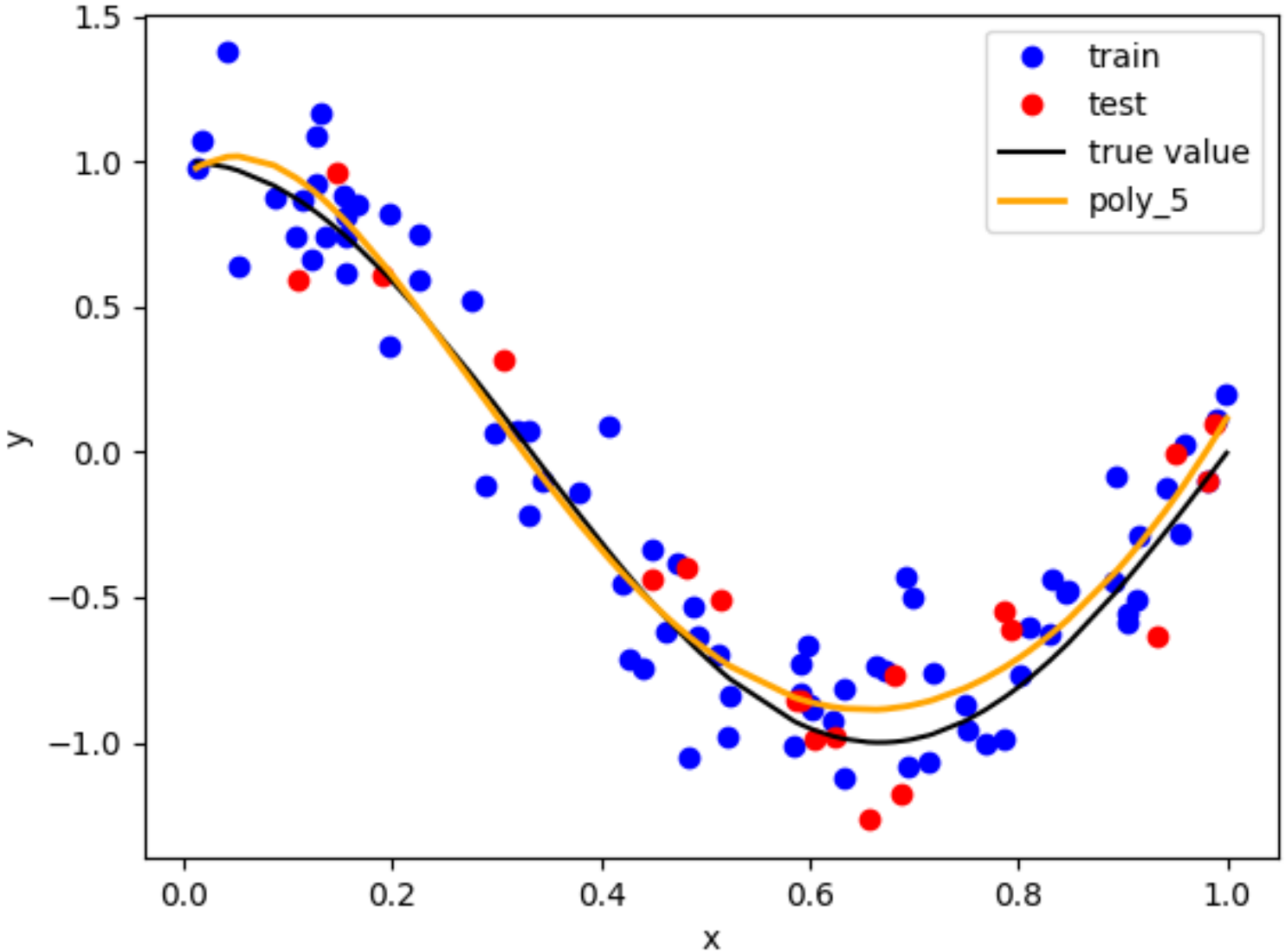
Let's perform higher degree polynomial regression

- allows for a curve instead of a line to be fitted
- e.g., quadratic equation: $y = ax^2 + bx + c$
- we'll fit both a good and poor curve

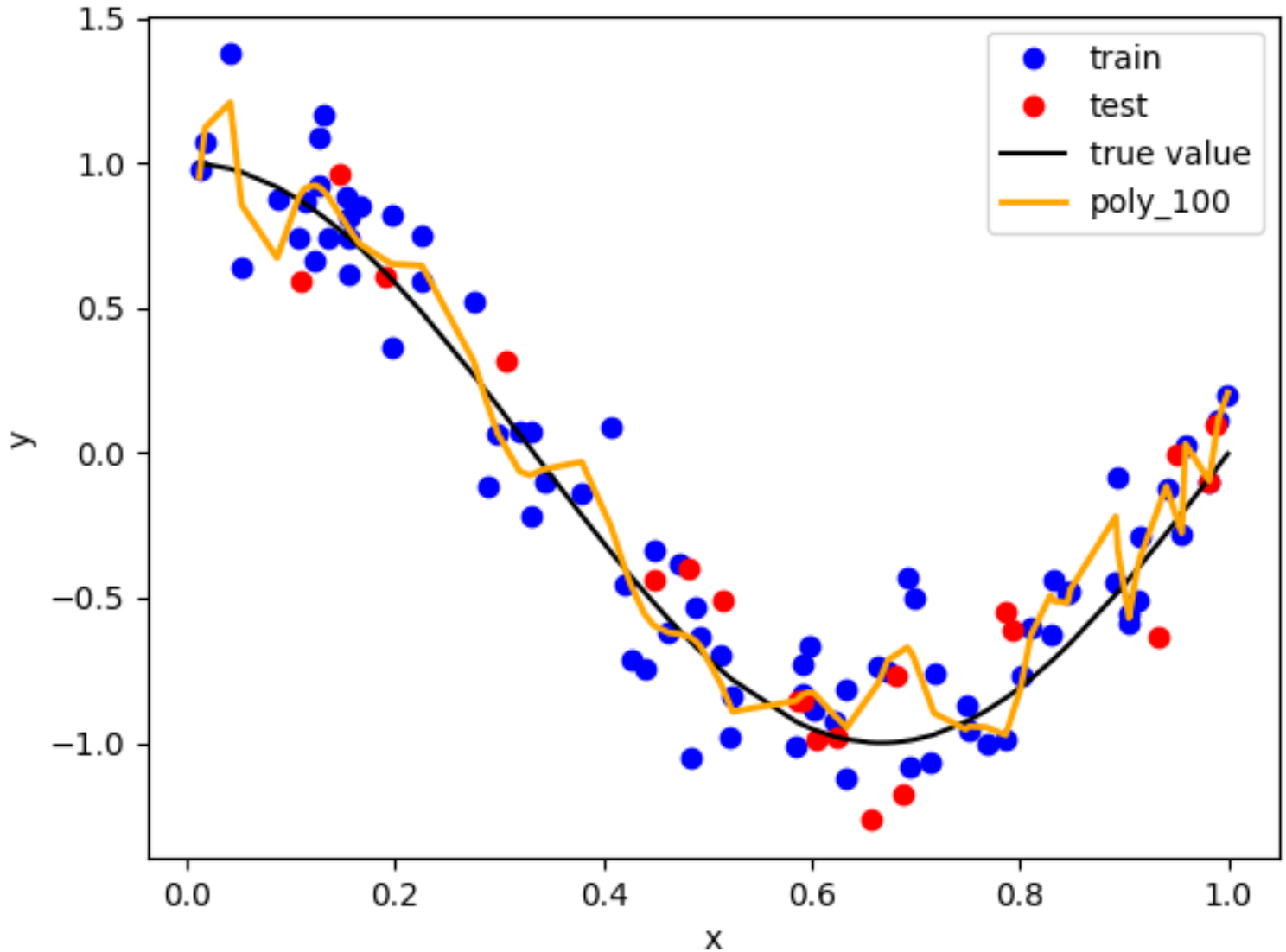
Fitted curves:

1. degree 5 (good): $y = ax^5 + bx^4 + cx^3 + dx^2 + ex + f$
2. degree 100 (bad)

Polynomial regression - degree 5



Polynomial regression - degree 100



Analysis summary

Degree	σ_{train}	σ_{test}
1	0.237980438732	0.310466053348
5	0.0350559930311	0.0402136253722
100	0.0234195789357	211.356355753

Table: Summary of MSE for various degrees of polynomial regression

There's a clear indication of under- and overfitting occurring

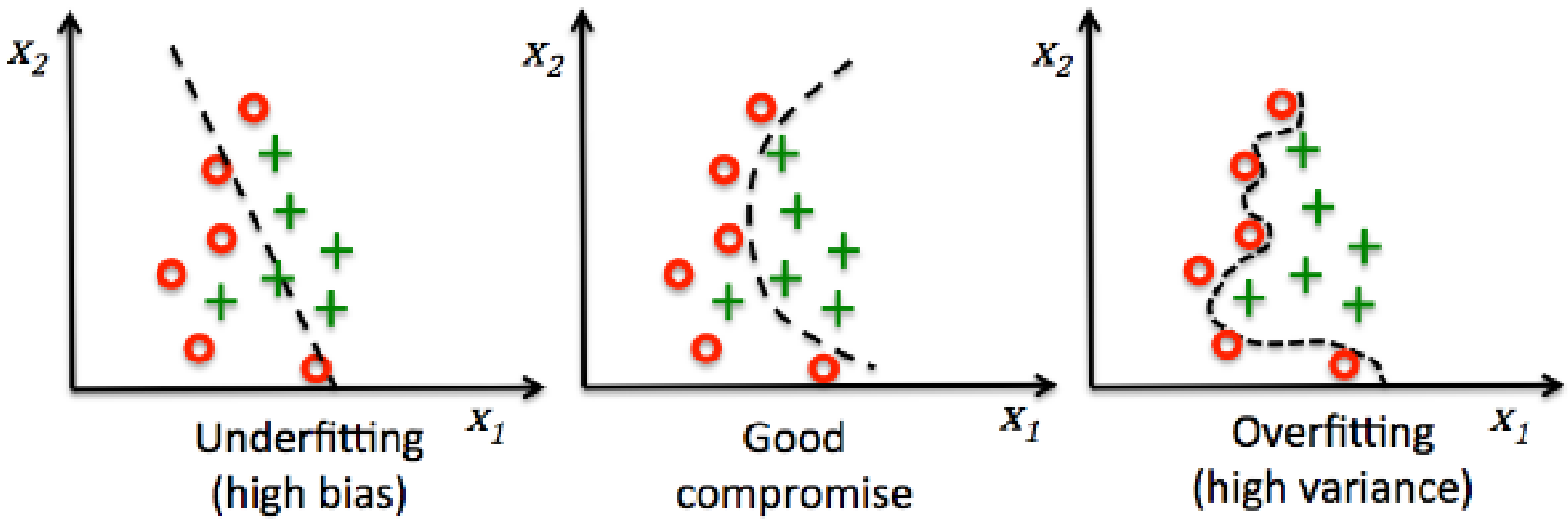
- underfitting: linear regression fails to learn cosine
- overfitting: 100th degree polynomial memorizes training data
  - but performs poorly on unseen data (testing)

Overfitting #3

What is overfitting?

- occurs when the ML algorithm learns a function that fits too closely to a limited set of data points
- predictions on unseen data will be biased to **training data**
  - increased error for **testing data** during **evaluation**

Example: draw a line that best splits 'o's from '+'s below



Back to our Titanic survival problem

At the end of the last lecture, we had

- data mined the Titanic dataset
- prepared the dataset for a ML algoirthm
  1. removed features with a low number of examples
  2. removed passengers with missing data
  3. encoded categorical string features as numeric representations
  4. split data into training and testing datasets

The next step is to choose a ML algorithm

- first we need to decide on what type of algorithm to use
- should it be a **classifier** or **regressor**?

Dataset features

After preparing the dataset:

- features #1-7 are available
- input label/target is #8

Columns:

1. 'pclass' - passenger class (1, 2, or 3)
2. 'sex' - sex of passenger (male (1) or female (0))
3. 'age' - age of passenger in years (float)
4. 'sibsp' - number of siblings/spouses aboard (integer)
5. 'parch' - number of parents/children aboard (integer)
6. 'fare' - fare paid for ticket (float)
7. 'embarked': port of embarkation
   (Cherbourg (0); Queenstown (1); S = Southampton (2))
8. 'survived' - yes (1) or no (0)

Support vector machines (SVM)

SVMs work by trying to split data into groupings

- much like k-means clustering
- can be binary classes or multiclass

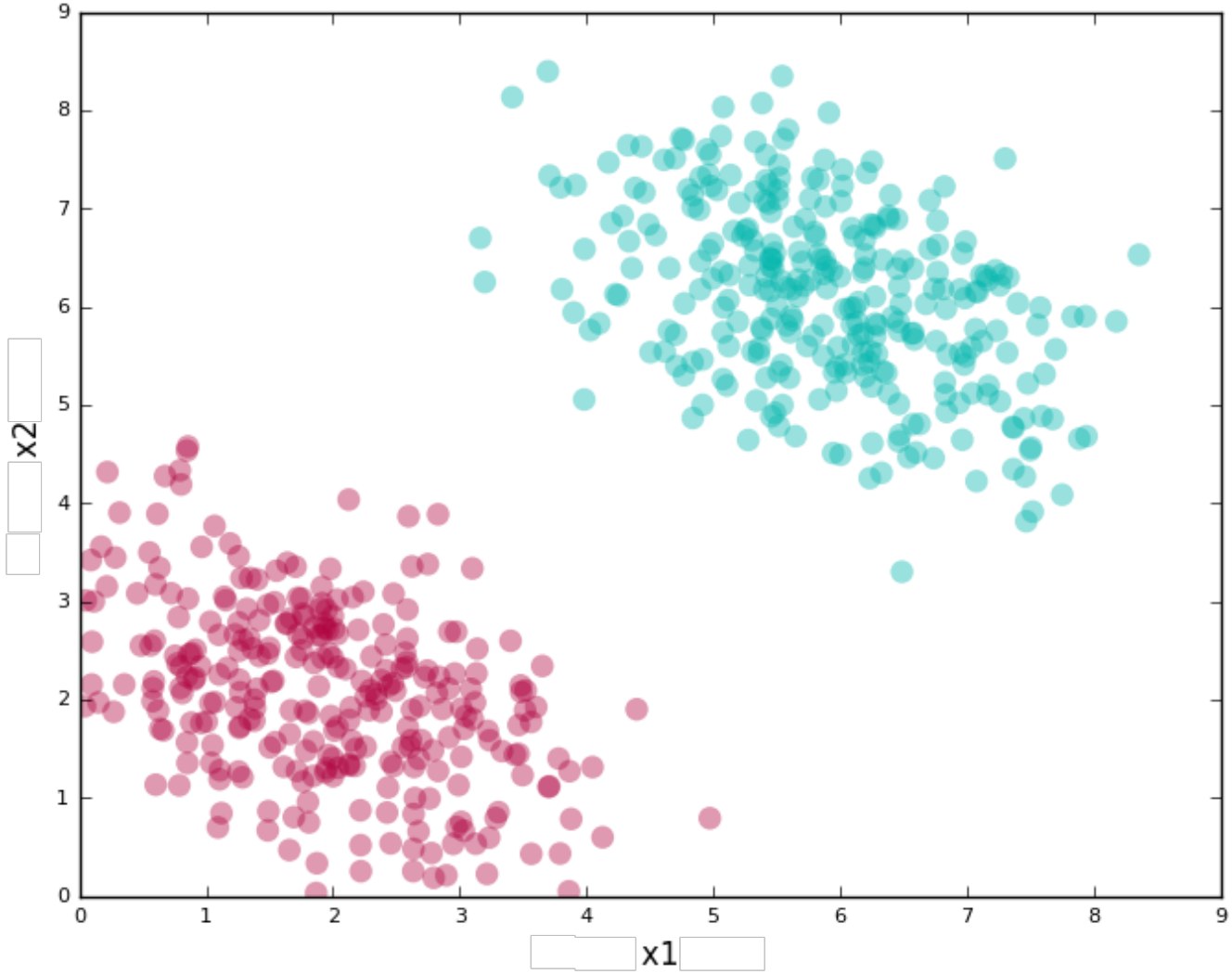
Simplified steps of an SVM:

1. find lines that correctly classify training data
2. from said lines, choose the one that has the greatest distance to closest data points
    - **why is this important?**

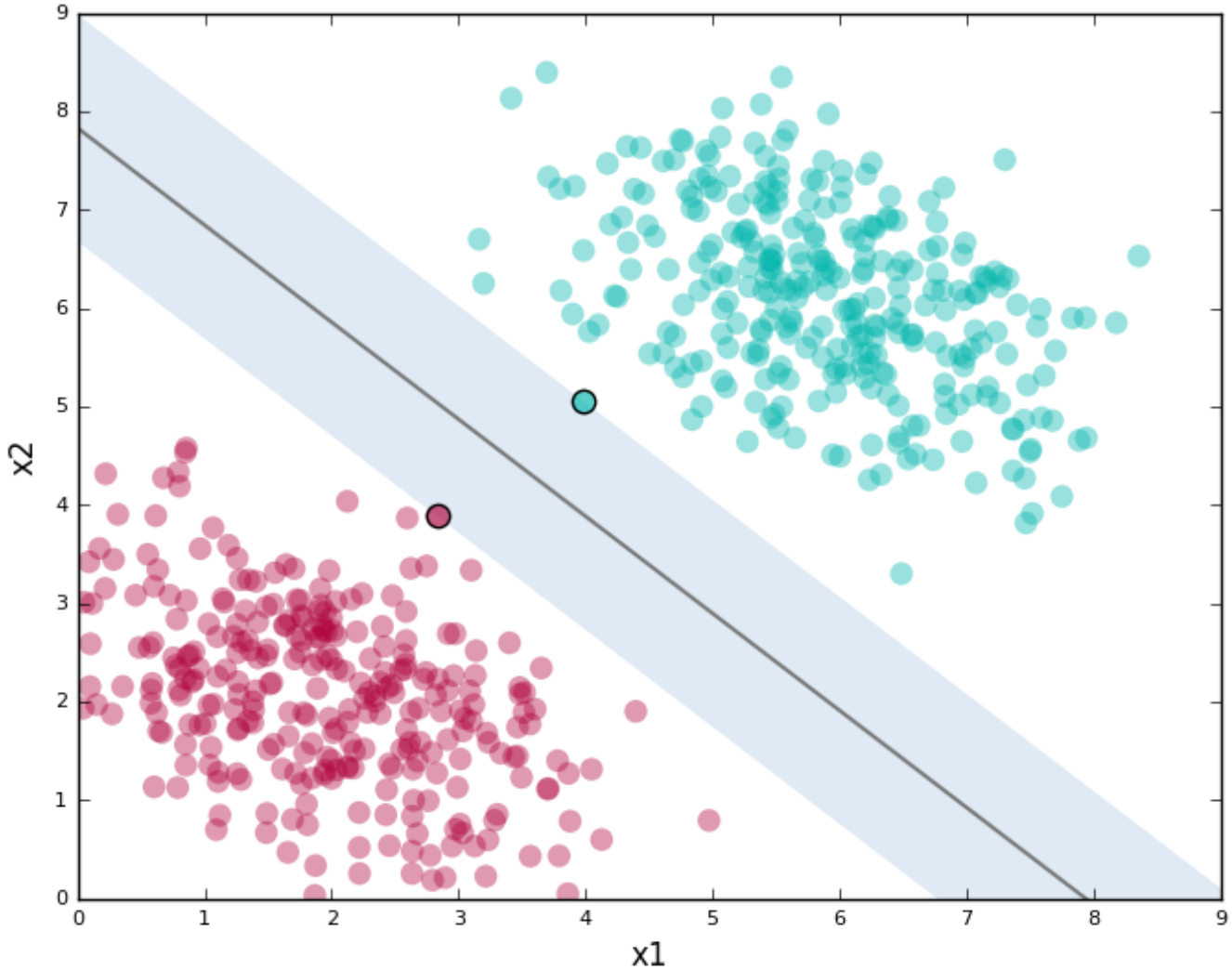
scikit-learn SVM classifier API

`http://scikit-learn.org/stable/modules/svm.html#classification`

SVM example



SVM example #2



The two highlighted points are the **support vectors**

SVMs #2

Support vectors are key to an SVM

- they define a region known as the **margin**
- in the previous example, it is the light-blue shaded regions

SVMs work to maximize the margin

- the larger the margin
- the less influence data variance has on a model
  - remember the bias vs. variance tradeoff?

Now that we know about SVMs, let's fit one to our data

```
from sklearn import model_selection,svm

X = data.drop(['survived'], axis=1).values
y = data['survived'].values
results = model_selection.train_test_split(X, y,
            test_size=0.2, shuffle = True)
X_train, X_test, y_train, y_test = results
# create classifier object
clf = svm.SVC()
# fit SVM to training data
clf.fit(X_train, y_train)
```

Notice that we keep calling the *fit*() method constantly?

- ML algorithms share a class inheritance in scikit-learn

Predictions with the SVM classifier

To make predictions with the learned SVM model

- pass in a list of model input exams
- in our case, we have bother training and testing data

```
clf = svm.SVC()
clf.fit(X_train, y_train)
pred_train = clf.predict(X_train)
pred_test = clf.predict(X_test)
print(pred_test[:10])
# prints: [0 0 0 0 0 0 1 1 0 1]
```

Model evaluation

We can also apply our MSE metric to the predictions

```
from sklearn.metrics import mean_squared_error

train_err = mean_squared_error(y_train,pred_train)
test_err = mean_squared_error(y_test,pred_test)
print("train_error:",train_err,"\ntest_error:",test_err)
#prints:
# train_error: 0.117505995204
# test_error: 0.320574162679
```

MSE isn't typically used for classification

- loss of information when comparing predictions to targets
- you don't know how close your prediction really is
- model predicts 0.65 that is rounded to 1 for the class label

Next time in COMP 364

Accuracy measures for classification

- ROC curves, precision, recall, etc.

Determining how important a feature is to the model's prediction

- 'feature importance'

Image analysis in Python with scikit-image

- **scikit-image API**
  `http://scikit-image.org/docs/dev/api/api.html`
- **scikit-image tutorials**
  `http://scikit-image.org/docs/dev/auto_examples/`