Supervised Machine Learning

Part One: Regression

DistrictDataLabs

Agenda

- Overview of Supervised Learning
- Regression Models (Algorithms)
- Model Evaluation
- Hands-on Lab

Supervised Learning

Definition

Supervised learning is the machine learning task of inferring a function from labeled training data. The training data consist of a set of training examples. In supervised learning, each example is a pair consisting of an input object (typically a vector) and a desired output value (also called the supervisory signal). A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples.

https://en.wikipedia.org/wiki/Supervised_learning

Important Points

1. Labeled training data
2. Desired output
3. Produces an inferred function
4. Used for novel examples

Approaches

1. Classification
2. Regression

Regression Models

Regression Models

- Supervised learning algorithms that estimate the relationship among variables.
- Focus is on the relationship between a dependent variable (target) and 1(+) independent variables (predictor)
- Does the dependent variable change when the independent variable(s) change?
- Common algorithms
  - Generalized linear models

Generalized Linear Models

Linear Models

Linear Regression fits a linear model to the data by adjusting a set of coefficients *w* to minimize the residual sum of squares between observed responses & prediction.

(1) Linear model $y = X\beta + \epsilon$

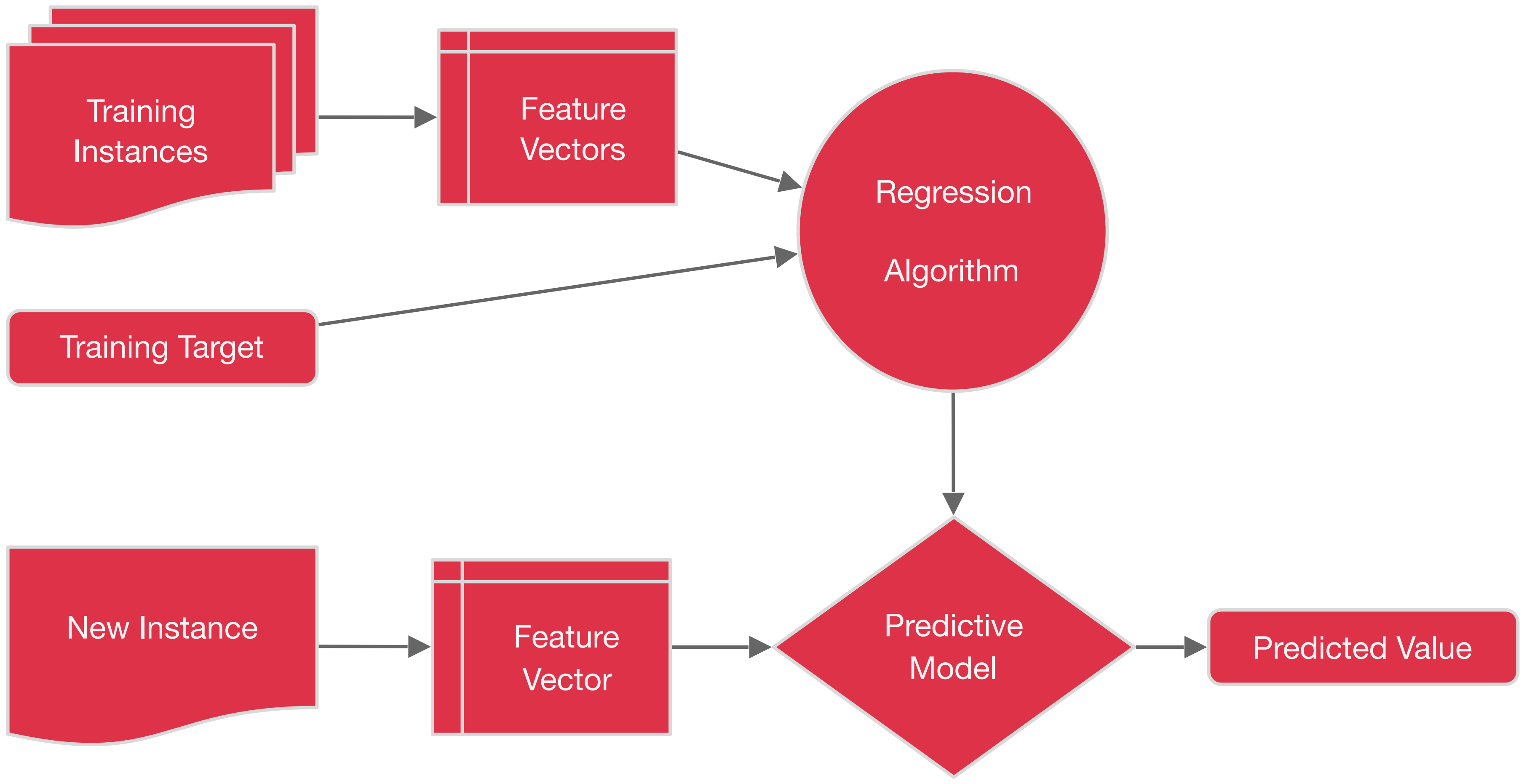
(2) Objective function $\min_{w} \sum (Xw - y)^2$

(3) Predictive model $\hat{y}(w,x) = w_0 + w_1 x_1 + \ldots + w_p x_p$

Notation:

- y is the observed value
- X is the input variables
- β is the set of coefficients
- ε is noise or randomness in observation
- w is the set of weights
- w_0 is the ability to adjust the plane in space
- $\hat{y}$ is the predicted value

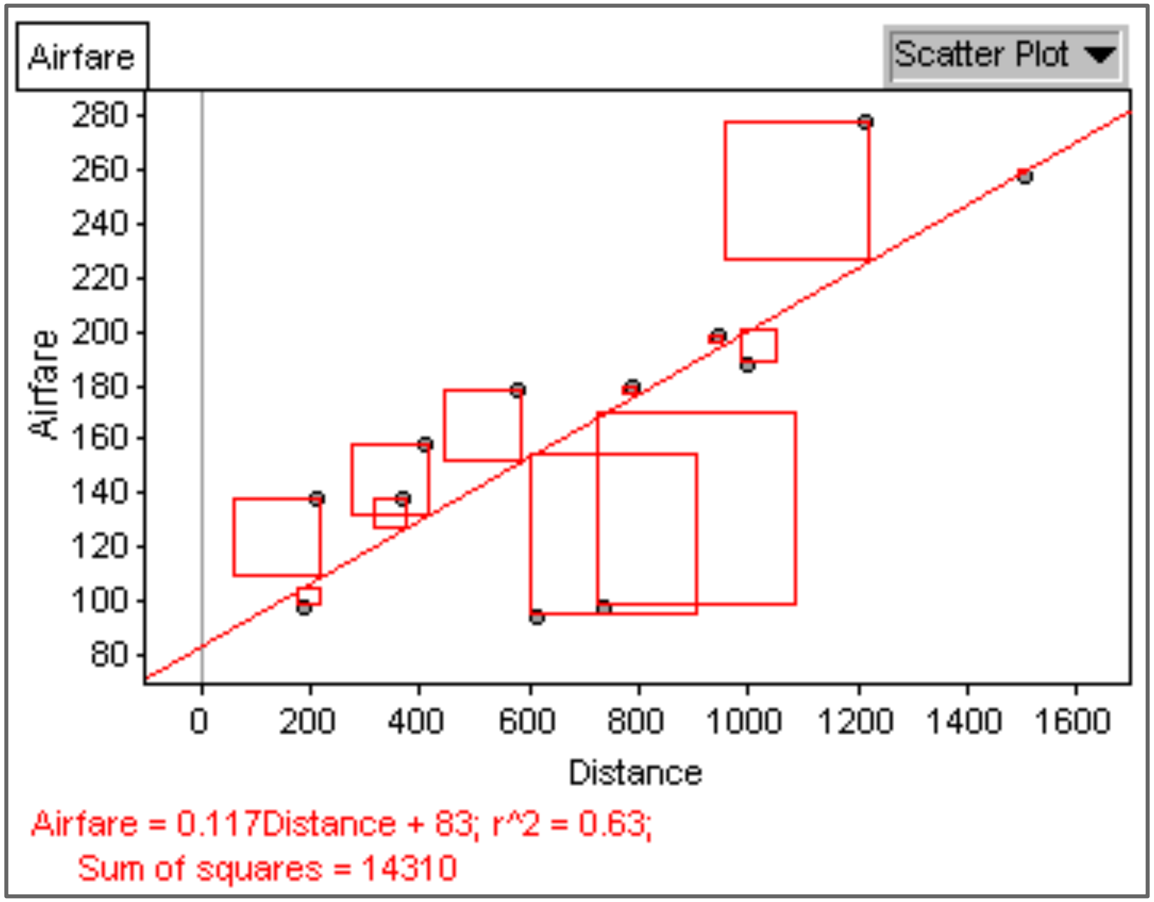
Regression Pipeline



DistrictDataLabs

Ordinary Least Squares

- Method for estimating unknown parameters in a linear regression model
- Keep adjusting parameters until minimum squared residuals (e.g. minimize some cost function).
- Relies on the independence of the model terms
- *multicollinearity*: two or more predictor variables in a multiple regression model are highly correlated, one can be linearly predicted from the others
- If this happens, the estimate becomes sensitive to error.

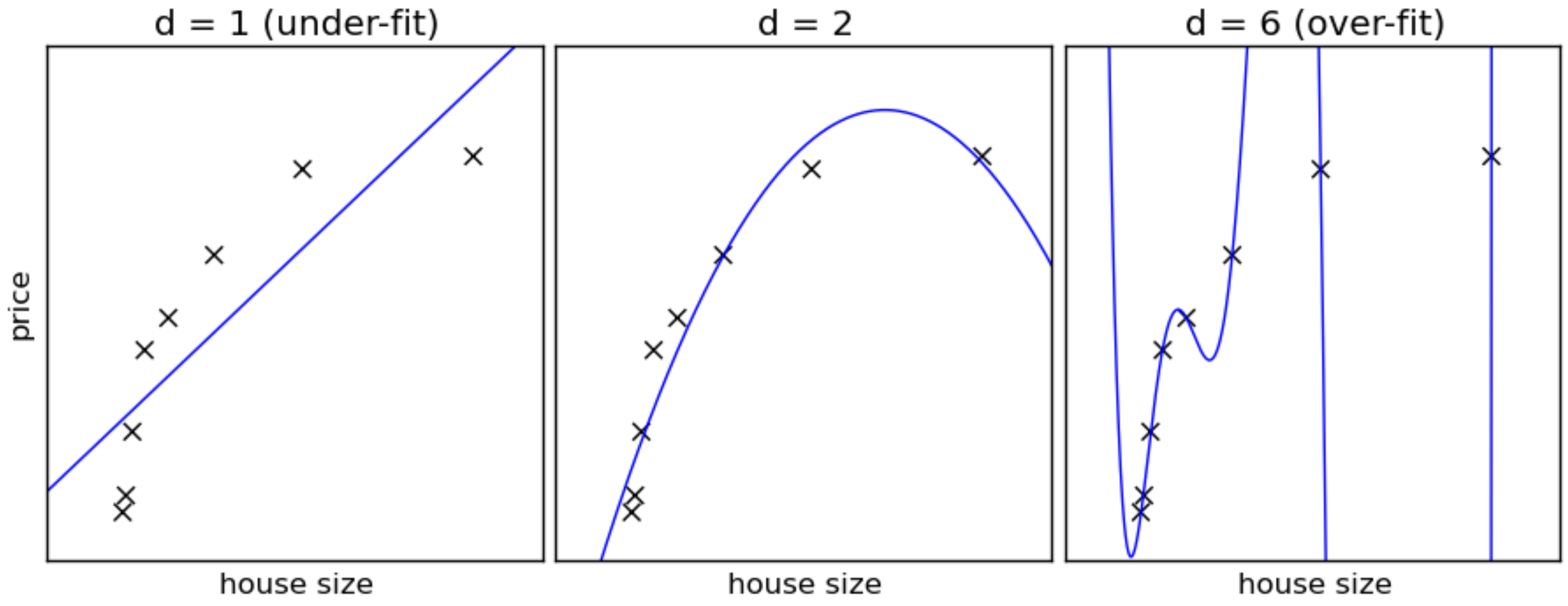


Airfare = 0.117Distance + 83; r^2 = 0.63;
Sum of squares = 14310

Simple Regression with OLS

```
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, r2_score


regr = linear_model.LinearRegression()
regr.fit(X_train, y_train)
LinearRegression(copy_X=True, fit_intercept=True, normalize=False)

print regr.coef_
[ -6.02985639e+01  -3.02367158e+11   3.02367158e+11   6.04734316e+11
   4.17860883e+00  -3.41060763e-02   2.03234971e+01   2.15758256e-01]

print regr.intercept_
76.9490920195

print mean_squared_error(y_test, regr.predict(X_test))
7.92744075579

regr.score(X_test, y_test)  # r2_score(y_test, regr.predict(X_test))
0.92521397739317868
```

What Can Go Wrong With Simple Linear Models?



Regularization

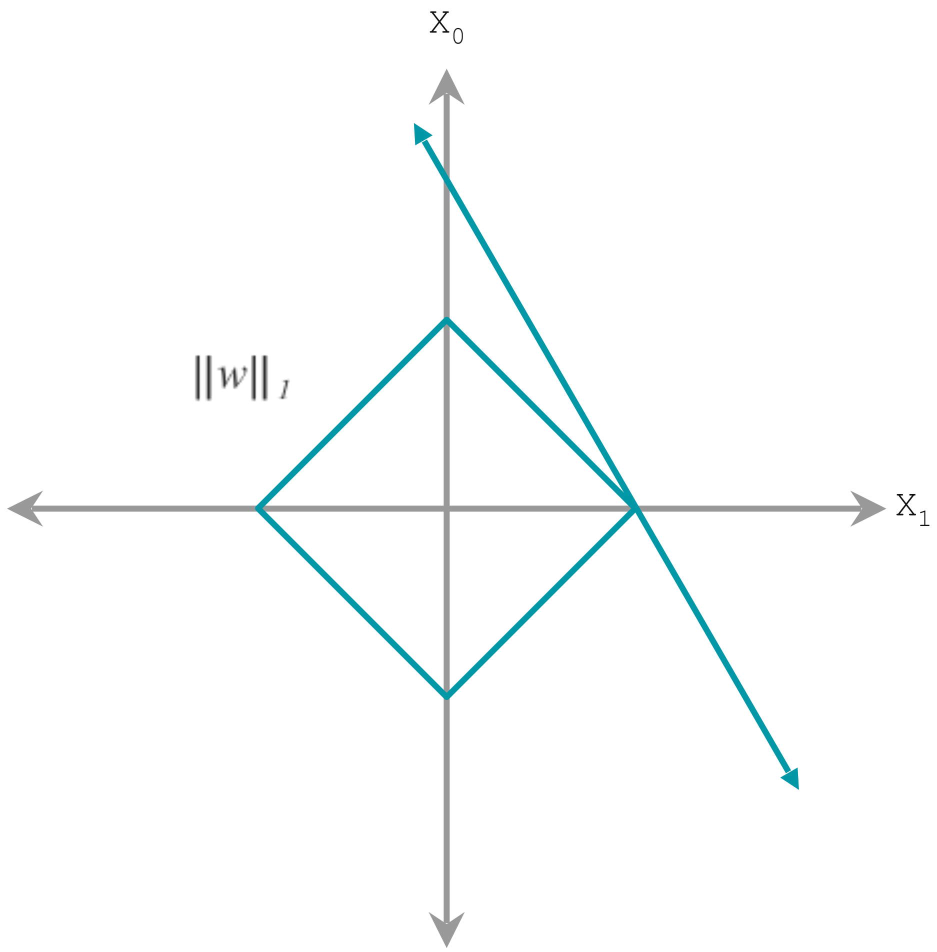
- As we increase the complexity of the model we reduce the bias but increase the variance of the model.
- Variance: the tendency for the model to fit to noise (randomness) -- overfit.
- Introduce a parameter to penalize complexity in the function being minimized.

Vector Norm

- Describes the length of the vector.
- L1: sum of the absolute values of components
- L2: euclidian distance from the origin
- L∞: maximal absolute value component

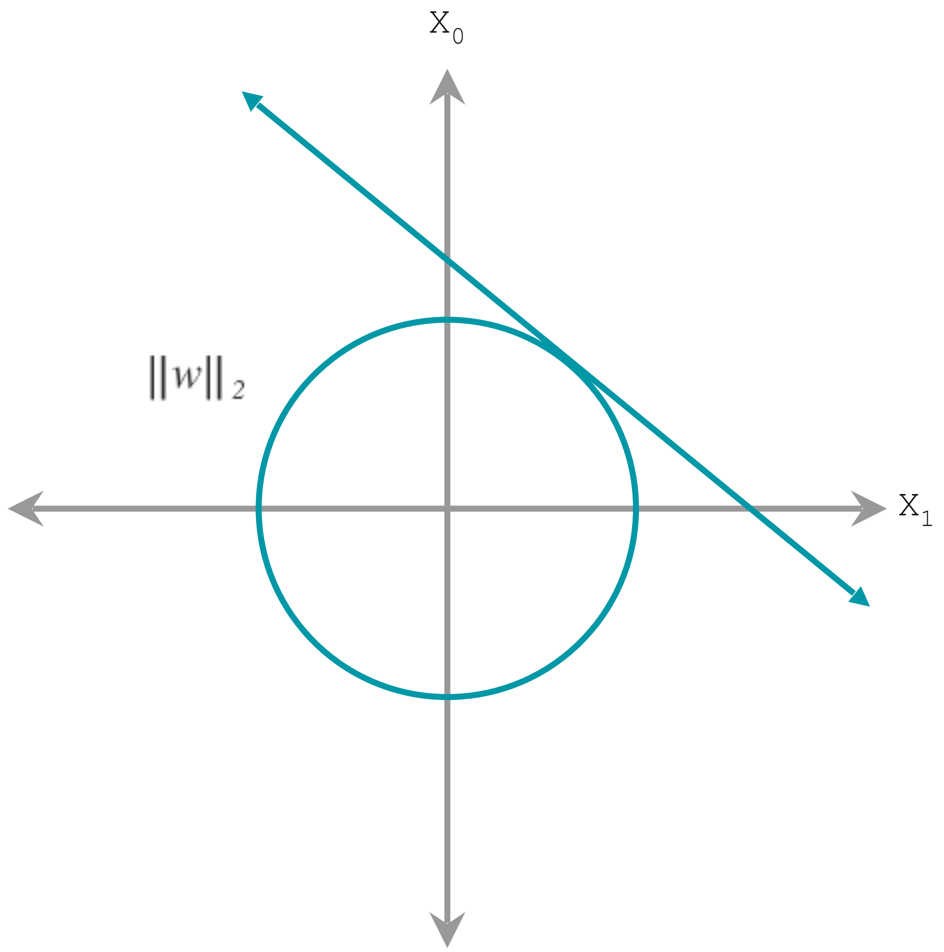
Vector Norm

```
import numpy as np
import numpy.linalg as la

vec = np.array([-10, 3, 3, -5, -3, -2, 1, 9, 3, 4, 6,
-8])
l1  = la.norm(vec, 1)
# 57.0

l2  = la.norm(vec, 2)
# 19.0525588833

lin = la.norm(vec, 'inf')
# 10
```



L1 Normalization

Possibility that a feature is eliminated by setting its coefficient equal to zero.



L2 Normalization

Features are kept balanced by minimizing the relative change of coefficients during learning.

Ridge Regression

- Prevent overfit/collinearity by penalizing the size of coefficients - minimize the penalized residual sum of squares:
- Said another way, shrink the coefficients to zero.

$$\min_{w} \sum (Xw-y)^2+\alpha \sum w^2$$

- Where $\alpha > 0$ is complexity parameter that controls shrinkage.

  The larger α, the more robust the model to collinearity.
- Alpha influences the bias/variance tradeoff: the larger the ridge alpha, the higher the bias and the lower the variance.

Ridge Regression

```
clf = linear_model.Ridge(alpha=0.5)
clf.fit(X_train, y_train)

Ridge(alpha=0.5, copy_X=True, fit_intercept=True,
      max_iter=None, normalize=False,
      solver=‘auto', tol=0.001)

print mean_squared_error(y_test, clf.predict(X_test))
8.34260312032

clf.score(X_test, y_test)
0.9212974117655727 8
```

Choosing `alpha`

We can search for the best parameter using the `RidgeCV` which is a form of Grid Search, but uses a more efficient form of leave-one-out cross-validation.

```
import numpy as np
n_alphas = 200
alphas = np.logspace(-10, -2, n_alphas)
clf = linear_model.RidgeCV(alphas=alphas)
clf.fit(X_train, y_train)

print clf.alpha_
0.0010843659686896108

clf.score(X_test, y_test)
0.92542477512171173
```

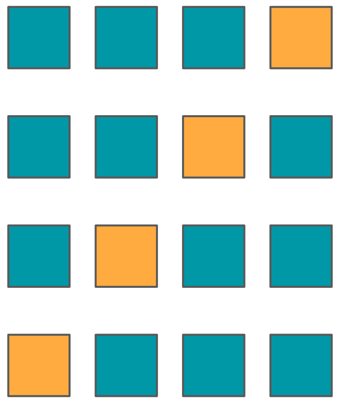
Model Evaluation

Cross-Validation and Evaluation

- In regressions we can determine how well the model fits by computing the mean square error and the coefficient of determination.
- MSE = np.mean((predicted-expected)**2)
- R^2 is a predictor of "goodness of fit" and is a value $\in [0,1]$ where 1 is perfect fit.

Cross-Validation and Evaluation

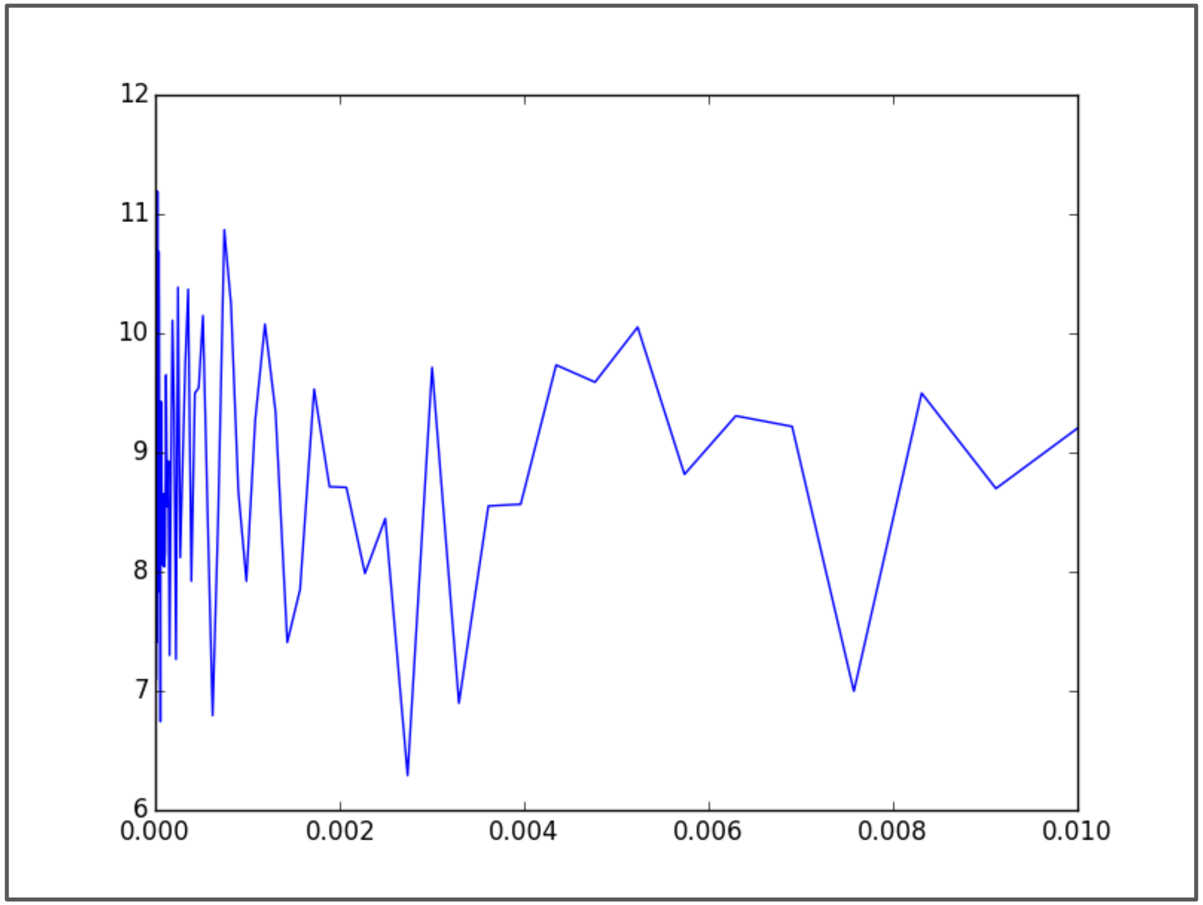
In order to prevent overfit and be assured of generalizability, cross-validation fits the model on a portion of the data set and evaluates it on an unseen portion of the data set. Shuffle data, split into a large train set and smaller test set. This can be done K=12 times, and scores averaged.



Error As a Function of Alpha

```
clf = linear_model.Ridge(fit_intercept=False)
errors = []


for alpha in alphas:
    splits = tts(dataset.data, dataset.target('Y1'), test_size=0.2)
    X_train, X_test, y_train, y_test = splits
    clf.set_params(alpha=alpha)
    clf.fit(X_train, y_train)
    error = mean_squared_error(y_test, clf.predict(X_test))
    errors.append(error)

axe = plt.gca()
axe.plot(alphas, errors)
plt.show()
```

Resulting Plot



How to pick the right parameters?

Search/Tuning

Search Requires

- Estimator
- Parameter Space
- Method for sampling
- Cross validation scheme
- A score function

Search Types

- Exhaustive
- Randomized
- Parallel
- Leave One Out
- Model Specific

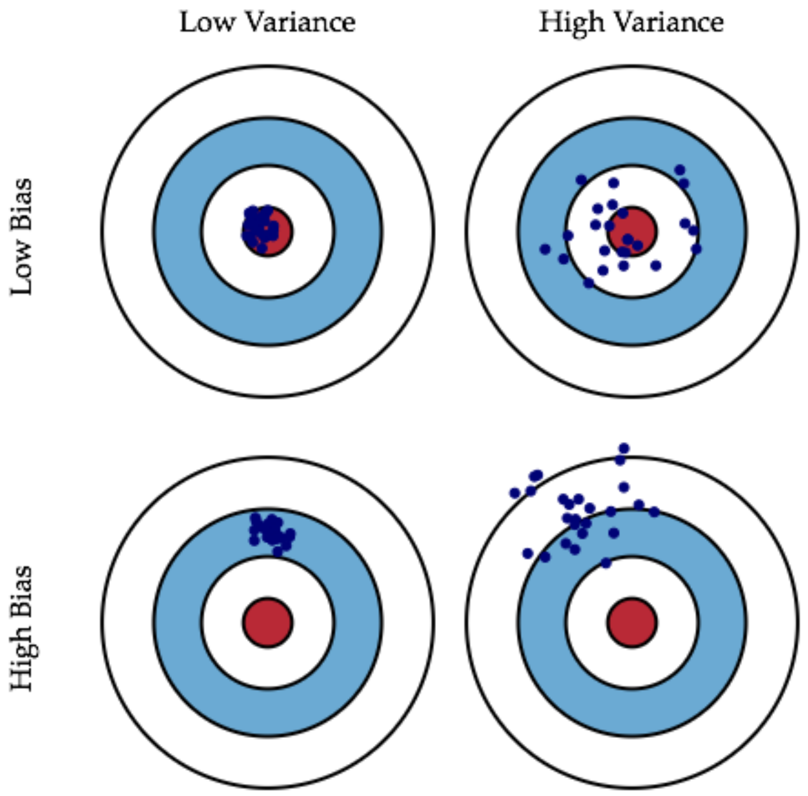
Creating an Exhaustive Grid Search for a Classifier

```
from sklearn.svm import SVC
from sklearn.grid_search import GridSearchCV

params = [
  {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
  {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel':
['rbf']},
]

estimator = GridSearchCV(SVC(), params)
estimator.fit(dataset.data, dataset.target)
```

Error: Bias vs Variance

Bias: the difference between expected (average) prediction of the model and the correct value.

Variance: how the predictions for a given point vary between different realizations for the model.



http://scott.fortmann-roe.com/docs/BiasVariance.html

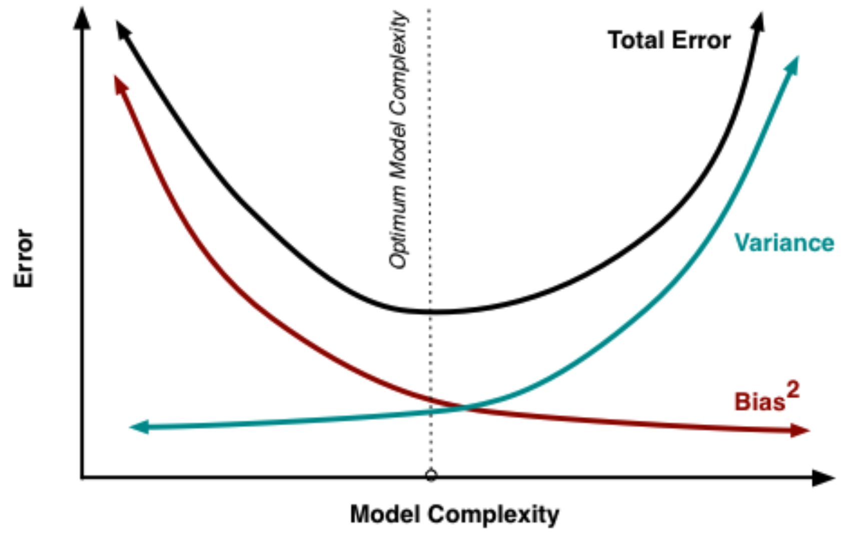
Bias vs. Variance Trade-Off

Related to model complexity:

The more parameters added to the model (the more complex), Bias is reduced, and variance increased.

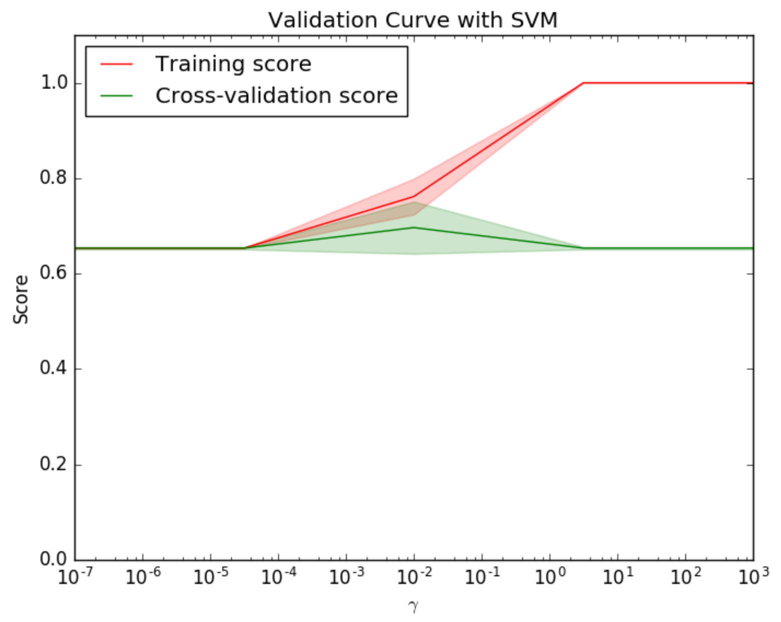
Sources of complexity:

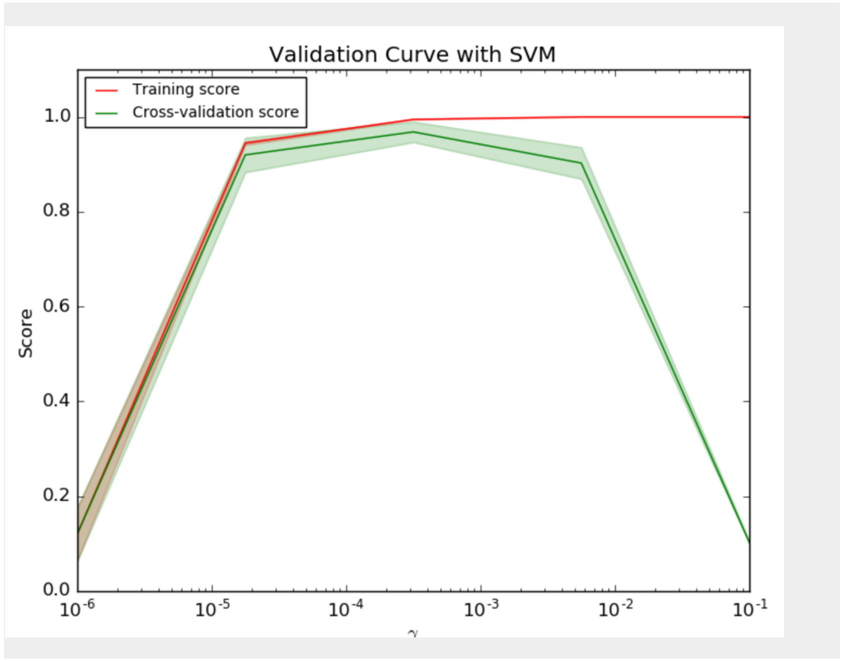
- k (nearest neighbors)
- epochs (neural nets)
- # of features
- learning rate



http://scott.fortmann-roe.com/docs/BiasVariance.html

Visual Parameter Tuning





Visual Parameter Tuning

- 2 different datasets: tic-tac-toe (left) & digits (right)
- Training vs. validation scores of SVM
- Different values of the kernel parameter gamma
- Things to look for:
  - Training score and validation score both low => Underfit
  - Training score high and validation score low => Overfit

Lasso

- Reducing bias is one thing, but what if the coefficients are very sparse? E.g. the more dimensions we add, the more *space* goes into the model.
- Lasso prefers fewer parameters attempting to reduce the number of variables the solution depends on.

$$\min_{w} \frac{1}{2n_{samples}} \left(\sum (Xw - y)^2 \right) + \alpha \|w\|_1$$

- The term $\alpha\|w\|_1$ is the L1 norm, whereas in ridge we used the L2 norm, $\alpha\|w\|_2{}^2$.
- See also Least Angle Regression (LARS) as similar.
- Can also use `LassoCV` and `LassoLarsCV`

Lasso Regression

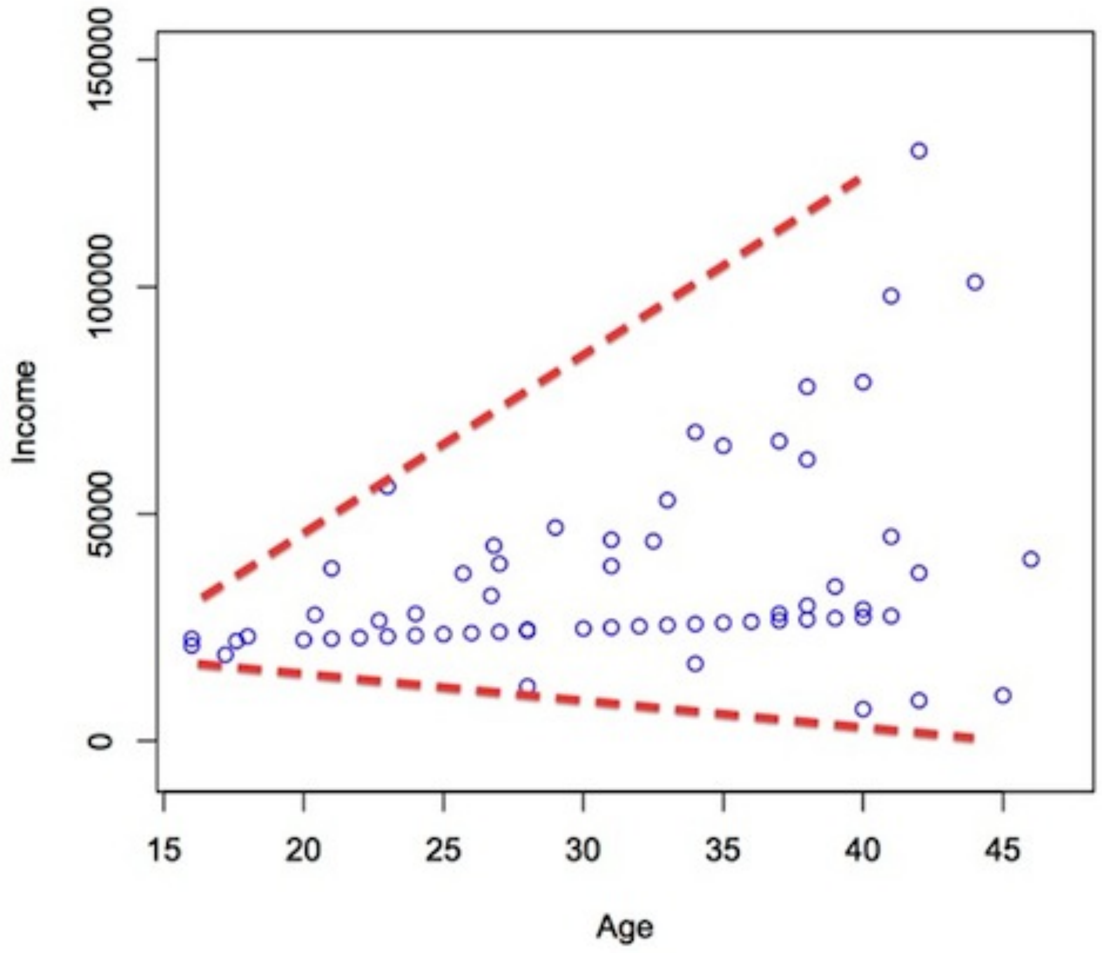
```
clf = linear_model.Lasso(alpha=0.5)
clf.fit(X_train, y_train)


Lasso(alpha=0.5, copy_X=True, fit_intercept=True,
      max_iter=1000, normalize=False, positive=False,
      precompute='auto', tol=0.0001, warm_start=False)

print mean_squared_error(y_test, clf.predict(X_test))
18.84667821

clf.score(X_test, y_test)
0.828704917633341947
```

Instance Variance

Heteroscedasticity: variability of variable is unequal along range of predicted values.

Homoscedasticity: variance is equal along prediction (assumed in most models).

Instance Variance



And More Models

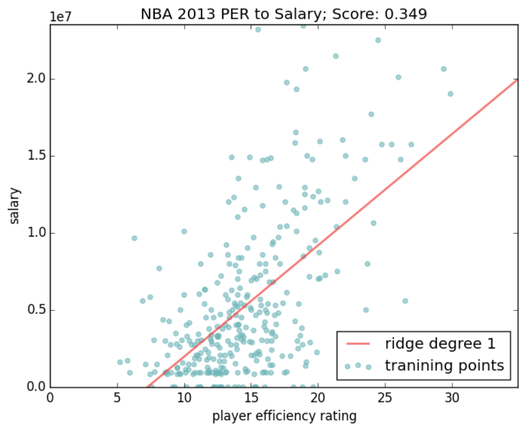
Listed only from the Documentation (not API):

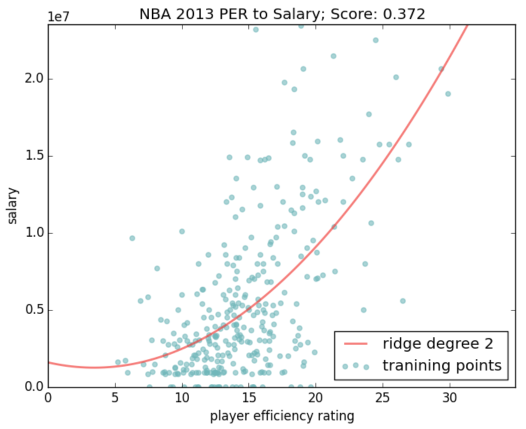
- ElasticNet
- Multi-Task Lasso
- Least Angle Regression
- LARS Lasso
- Orthogonal Matching Pursuit (OMP)
- Bayesian Regression
- Automatic Relevance Determination (ARD)
- Logistic Regression
- Stochastic Gradient Descent
- Perceptron
- Random Sample Consensus (RANSAC)

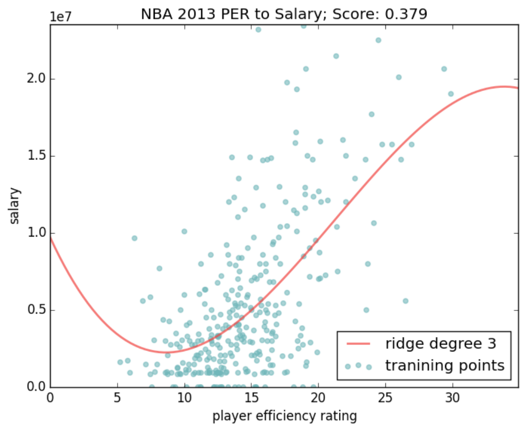
Polynomial Regression

In order to do higher order polynomial regression, we can use *linear models* trained on *nonlinear* functions of data!

- Speed of linear model computation
- Fit a wider range of data or functions
- But remember: polynomials aren't the only functions to fit







Polynomial Regression

The way this works is via *Pipelining*.
Consider the standard linear regression case:

$$\hat{y}(w,x) = w_0 + \sum_{i=1}^{n} w_i x_i$$

The quadratic case (polynomial degree = 2) is:

$$\hat{y}(w,v,x) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} v_i x_i^2$$

But this can just be seen as a new feature space:

$$z = [x_1, \ldots, x_n, x_1^2, \ldots, x_n^2]$$

And this feature space can be computed in a linear fashion.
We just need some way to add our 2nd degree dimensions.

Pipelined Model

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

model = make_pipeline(PolynomialFeatures(2), linear_model.Ridge())
model.fit(X_train, y_train)

Pipeline(steps=[('polynomialfeatures',
    PolynomialFeatures(degree=2, include_bias=True,
                       interaction_only=False)),
   (‘ridge',
    Ridge(alpha=1.0, copy_X=True, fit_intercept=True,
          max_iter=None, normalize=False, solver='auto',
          tol=0.001))])

mean_squared_error(y_test, model.predict(X_test))
3.1498887586451594

model.score(X_test, y_test)
0.9709057634510810 4
```

Pipelines (Steps)

`sklearn.pipeline.Pipeline`

- Sequentially apply *repeatable* transformations to final estimator that can be validated at every step.
- Each step (except for the last) must implement `Transformer`, e.g. `fit` and `transform` methods.
- Pipeline itself implements both methods of `Transformer` and `Estimator` interfaces.

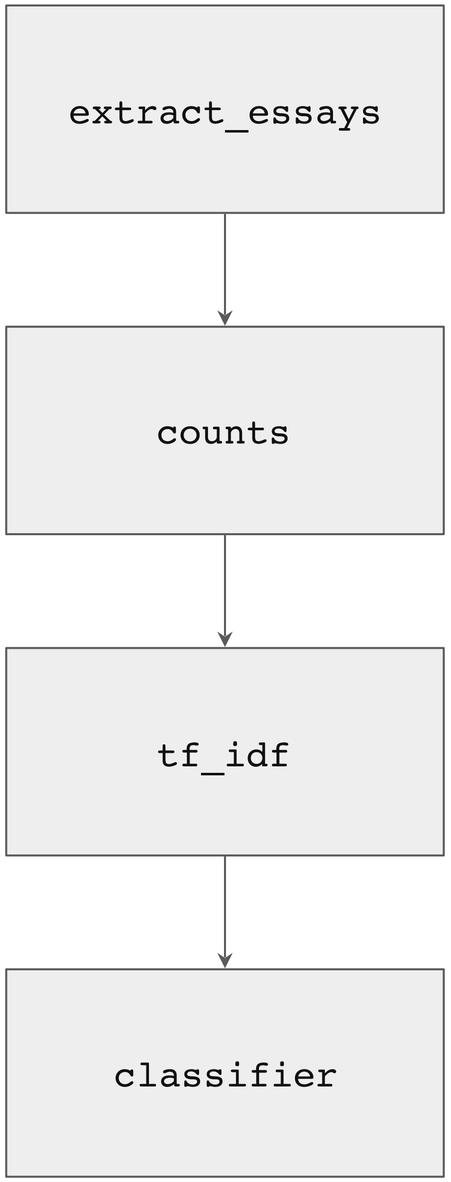
Transformers

```
class Transformer(Estimator):

    def transform(self, X):
        """Transforms the input data. """
        # transform ``X`` to ``X_prime``
        return X_prime

from sklearn import preprocessing

Xt = preprocessing.normalize(X) # Normalizer
Xt = preprocessing.scale(X)     # StandardScaler

imputer =Imputer(missing_values='Nan',
                 strategy='mean')
Xt = imputer.fit_transform(X)
```

Scikit-Learn Estimator API

```
class Estimator(object):

    def fit(self, X, y=None):
        """Fits estimator to data. """
        # set state of ``self``
        return self

    def predict(self, X):
        """Predict response of ``X``. """
        # compute predictions ``pred``
        return pred
```

```python
from sklearn.pipeline import Pipeline
from sklearn.cross_validation import KFold

pipeline = Pipeline([
    ('extract_essays', EssayExractor()),
    ('counts', CountVectorizer()),
    ('tf_idf', TfidfTransformer()),
    ('classifier', MultinomialNB())
])

scores = []
folds = KFold(

     n = dataset.data.shape[0], n_folds=12, shuffle=True

)

for tidx, cidx in folds:
    pipeline.fit(dataset.data[tidx], dataset.target[idx]
    score = pipeline.score(dataset.data[cidx],
dataset.target[cidx])
    scores.append(score)

print("Score: {}".format(np.mean(scores)))
```

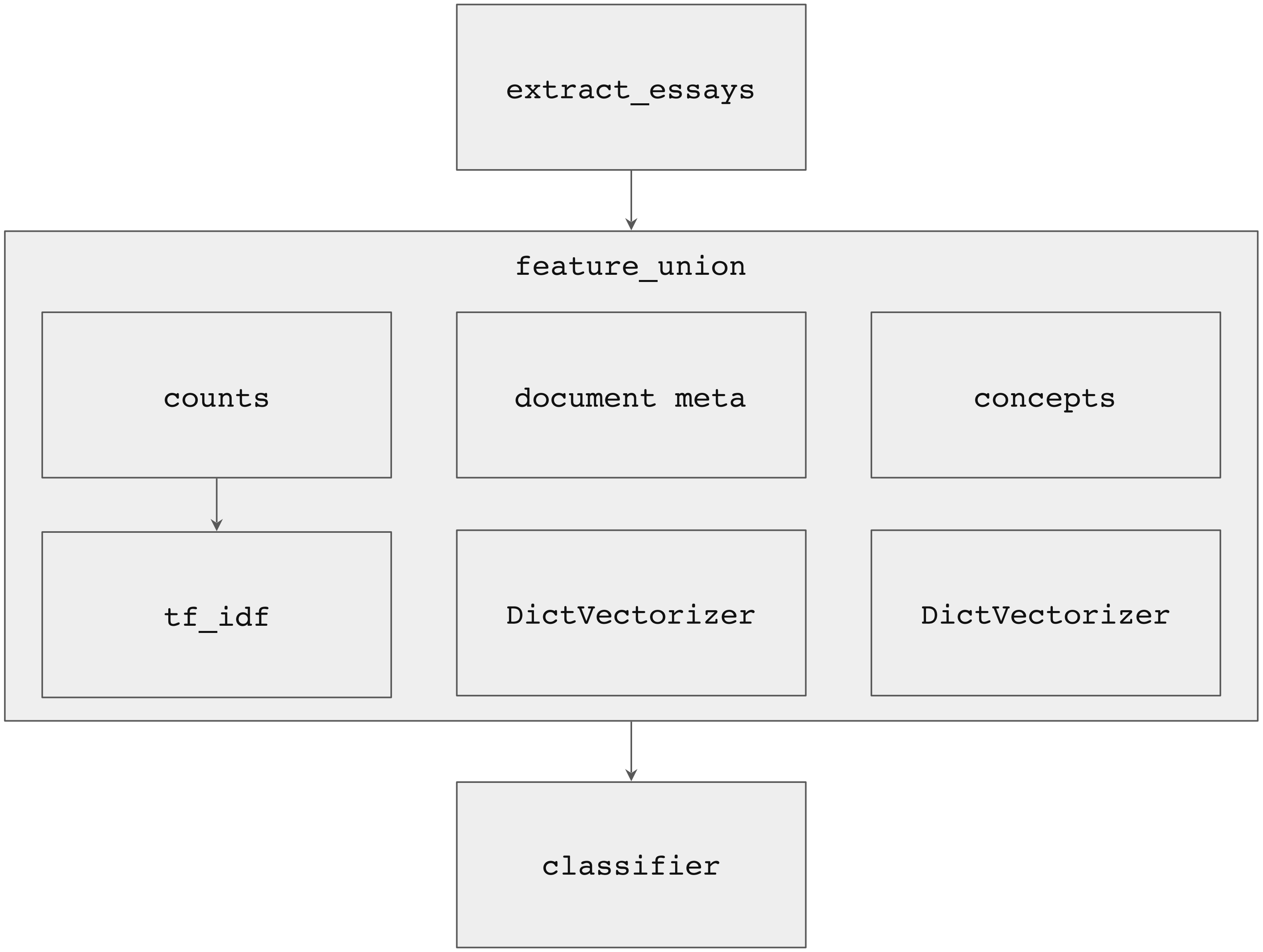


http://zacstewart.com/2014/08/05/pipelines-of-featureunions-of-pipelines.html

Pipelined Feature Extraction

The most common use for the Pipeline is to combine multiple feature extraction methodologies into a single, repeatable processing step.

- FeatureUnion
- SelectKBest
- TruncatedSVD
- DictVectorizer



Feature unions example from Zac’s post

Regression at Scale

What Makes Scikit-Learn Special

The Scikit-Learn API

Object-oriented interface centered around the concept of an Estimator:

“An estimator is any object that learns from data; it may be a classification, regression or clustering algorithm or a transformer that extracts/filters useful features from raw data.”

- Scikit-Learn Tutorial

The Scikit-Learn API

```
class Estimator(object):

    def fit(self, X, y=None):
        """
        Fits estimator to data.
        """
        # set state of self
        return self

    def predict(self, X):
        """
        Predict response of X
        """
        # compute predictions pred
        return pred
```

Buitinck, Lars, et al. "API design for machine learning software: experiences from the scikit-learn project." arXiv preprint arXiv:1309.0238 (2013).

The Scikit-Learn API

```python
class Transformer(Estimator):

    def transform(self, X):
        """
        Transforms the input data.
        """
        # transform X to X_prime
        return X_prime
```

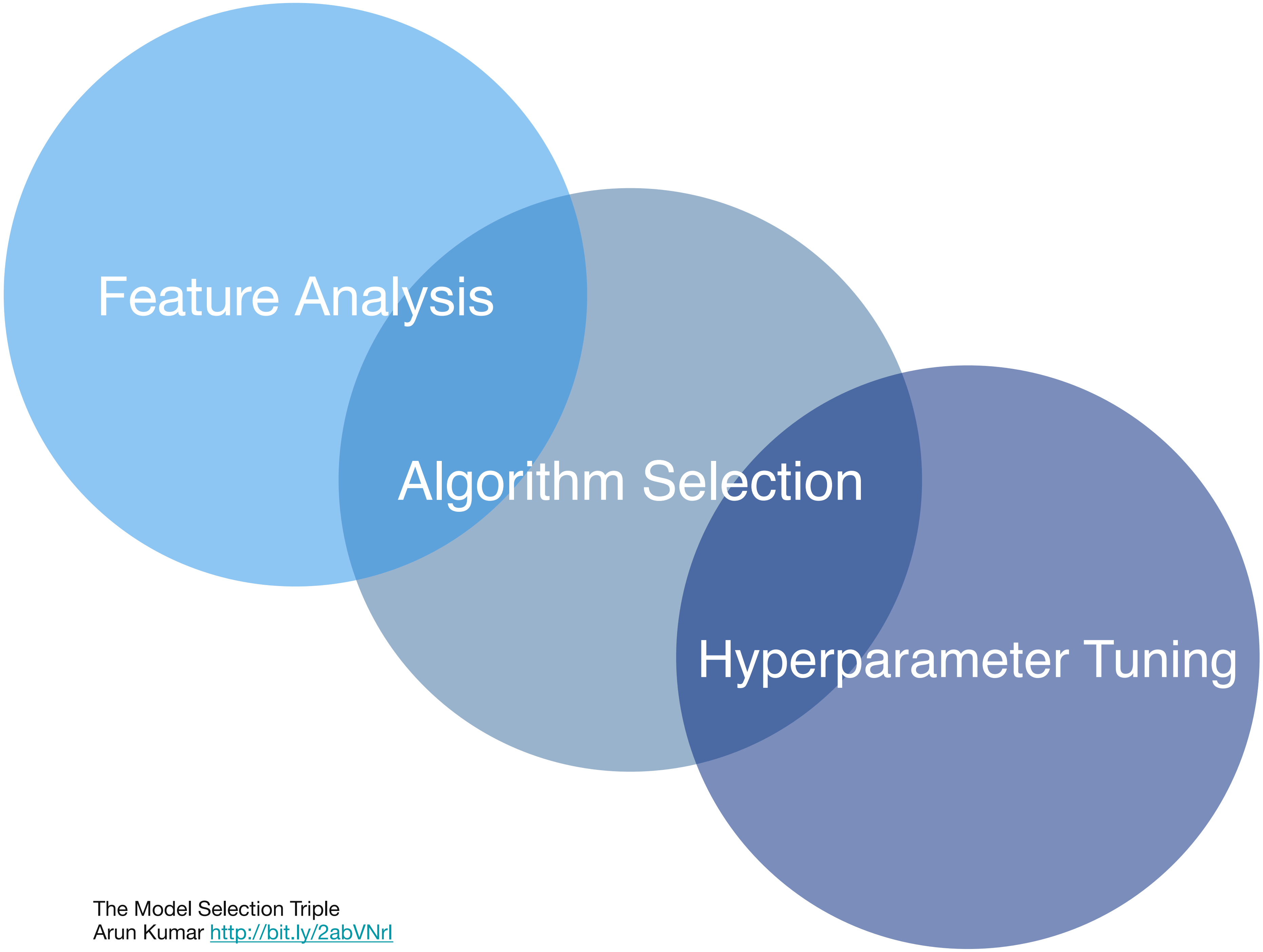
Buitinck, Lars, et al. "API design for machine learning software: experiences from the scikit-learn project." arXiv preprint arXiv:1309.0238 (2013).

The Scikit-Learn API

```python
class Pipeline(Transformer):

    @property
    def named_steps(self):
        """
        Returns a sequence of estimators
        """
        return self.steps

    @property
    def _final_estimator(self):
        """
        Terminating estimator
        """
        return self.steps[-1]
```

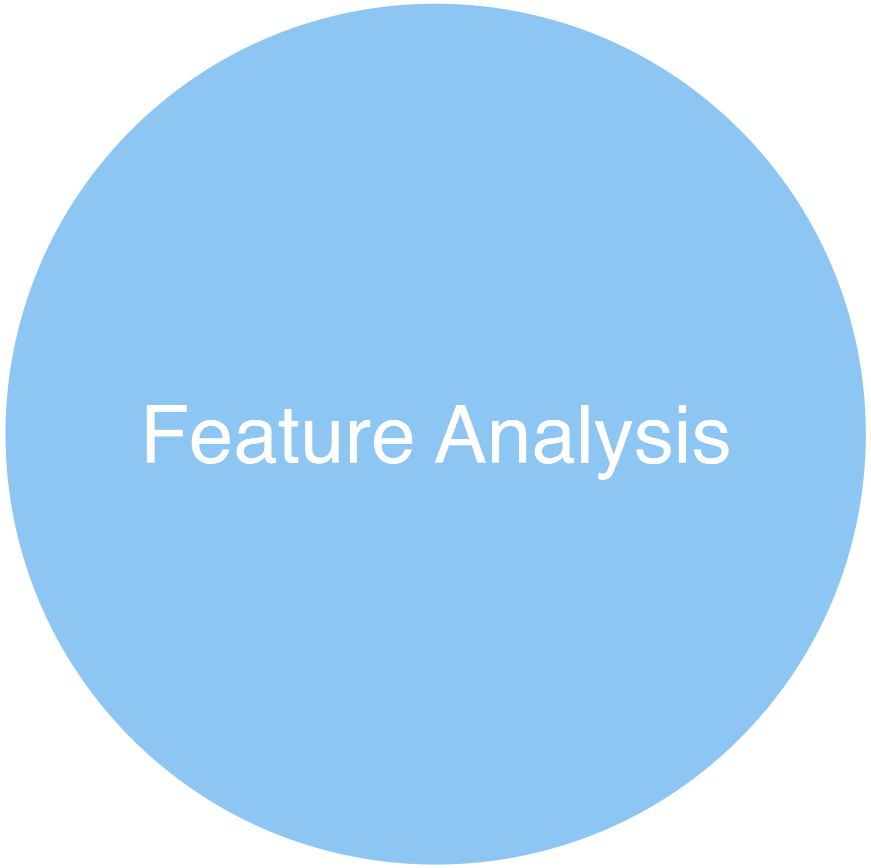
Buitinck, Lars, et al. "API design for machine learning software: experiences from the scikit-learn project." arXiv preprint arXiv:1309.0238 (2013).

The Model Selection Triple



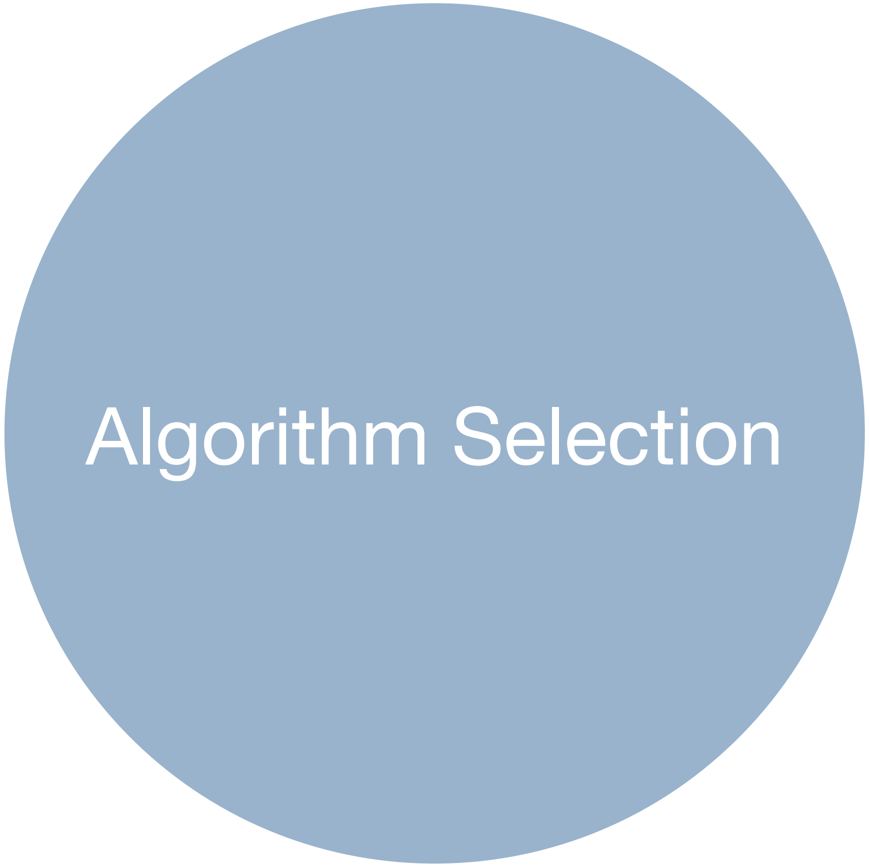
The Model Selection Triple
Arun Kumar http://bit.ly/2abVNrl

The Model Selection Triple



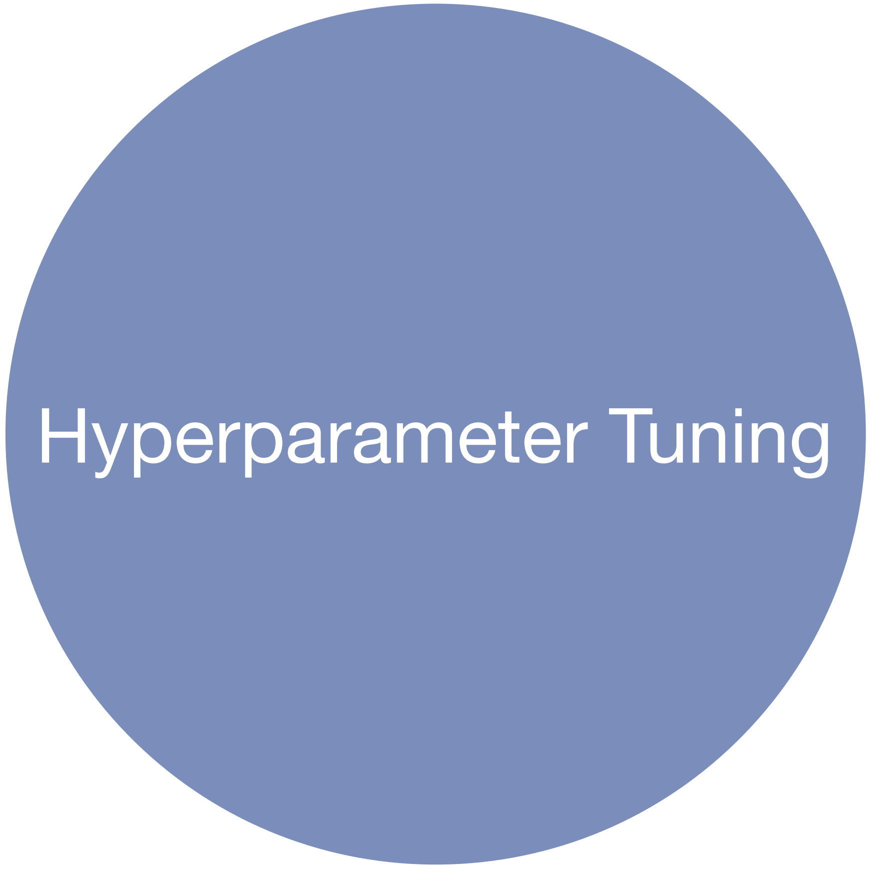
- Define a bounded, high dimensional feature space that can be effectively modeled.
- Transform and manipulate the space to make modeling easier.
- Extract a feature representation of each instance in the space.

The Model Selection Triple



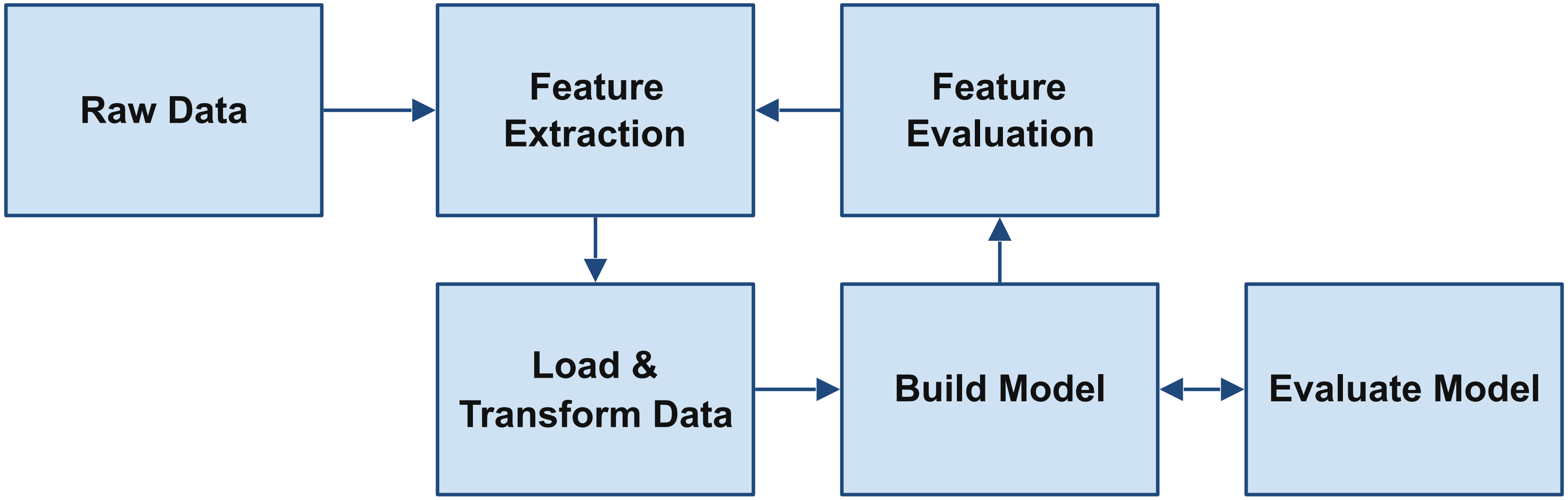
- Select a model family that best/correctly defines the relationship between the variables of interest.
- Define a model form that specifies exactly how features interact to make a prediction.
- Train a fitted model by optimizing internal parameters to the data.

The Model Selection Triple

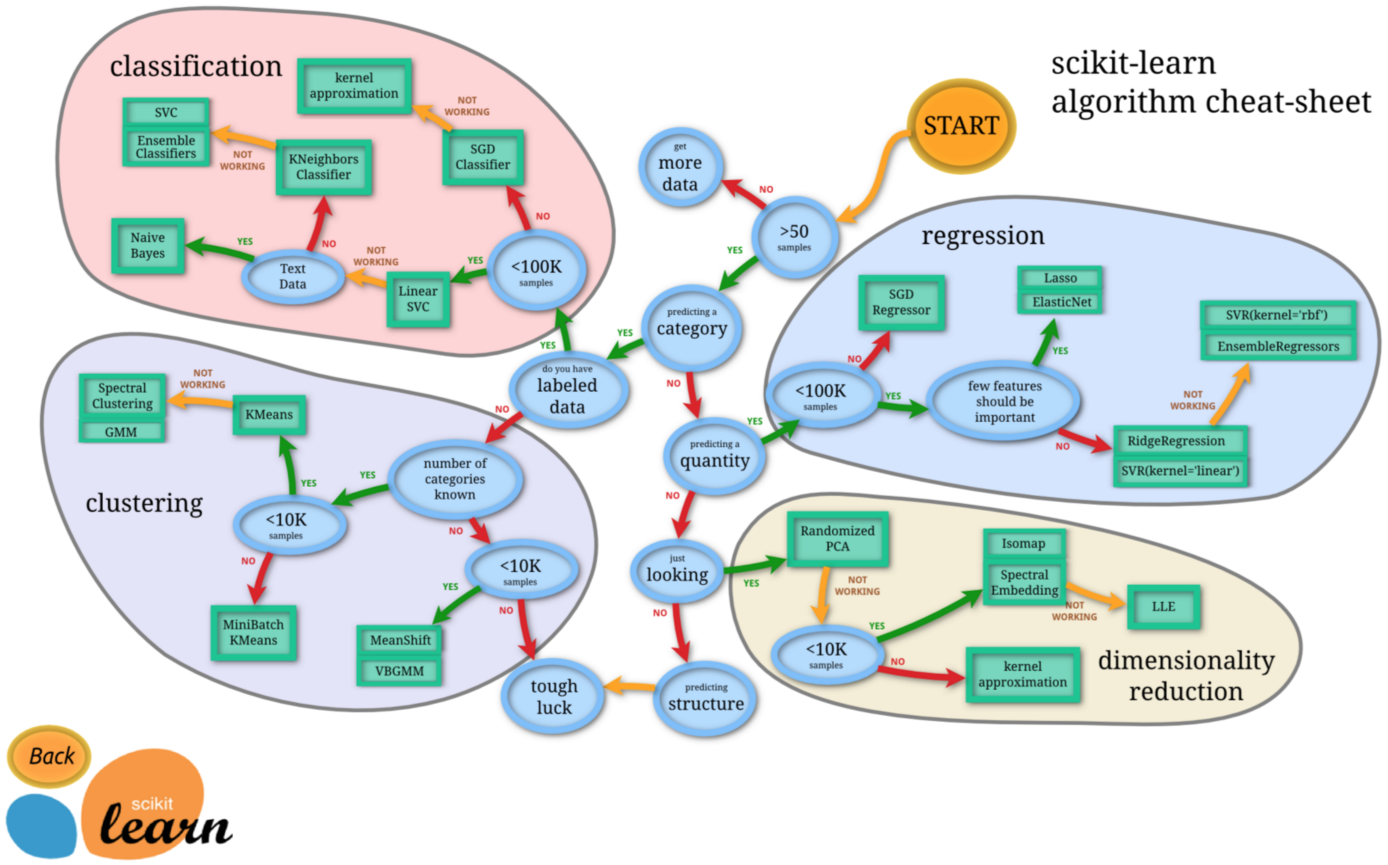


- Evaluate how the model form is interacting with the feature space.
- Identify hyperparameters (parameters that affect training or the prior, not prediction)
- Tune the fitting and prediction process by modifying these params.

Preliminary Workflow

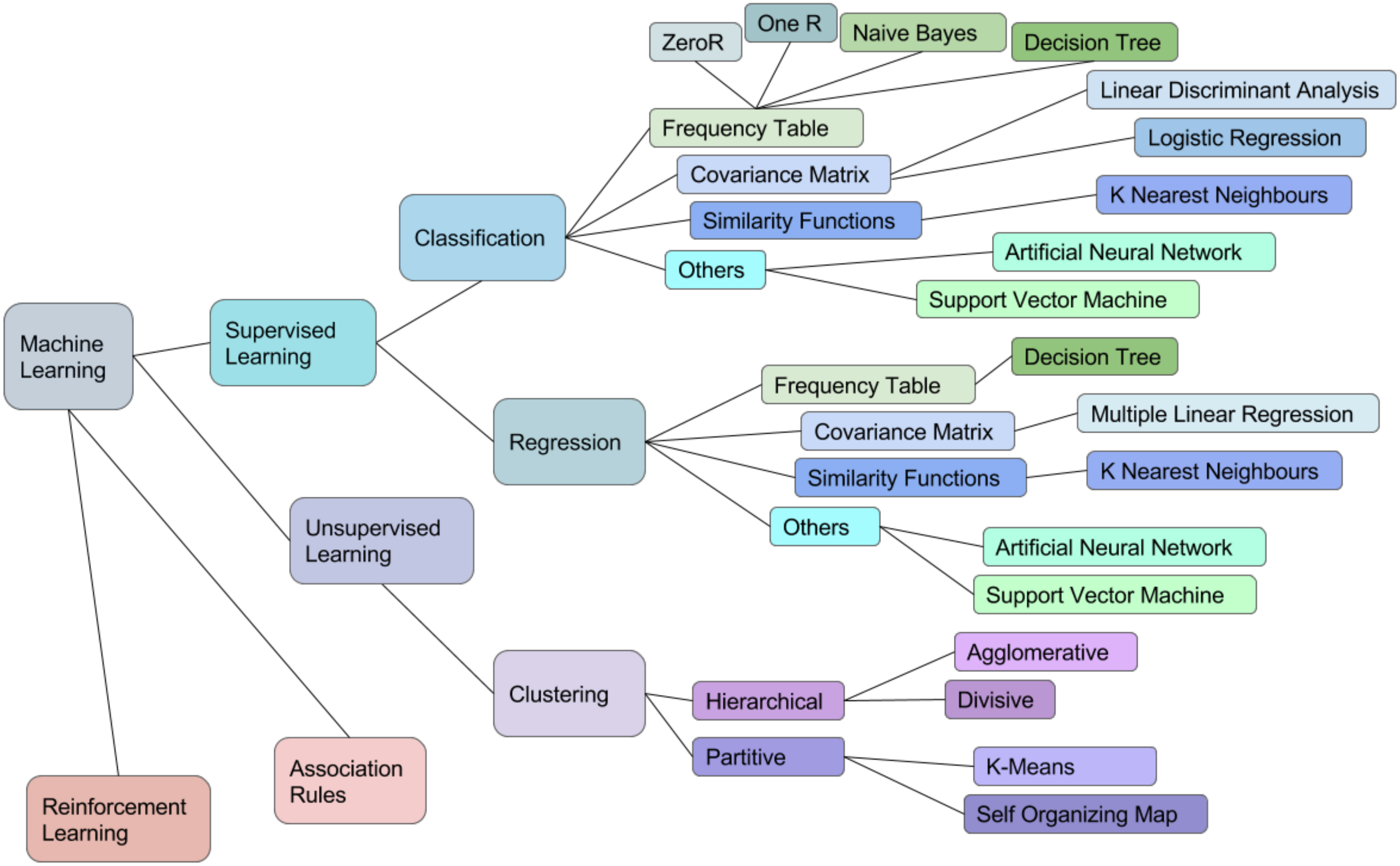


Choosing the Right Estimator



Machine Learning for Big Data

DistrictDataLabs


Machine Learning
Supervised Learning
Classification
Frequency Table
ZeroR
One R
Naive Bayes
Decision Tree
Covariance Matrix
Linear Discriminant Analysis
Logistic Regression
Similarity Functions
K Nearest Neighbours
Others
Artificial Neural Network
Support Vector Machine
Regression
Frequency Table
Decision Tree
Covariance Matrix
Multiple Linear Regression
Similarity Functions
K Nearest Neighbours
Others
Artificial Neural Network
Support Vector Machine
Unsupervised Learning
Clustering
Hierarchical
Agglomerative
Divisive
Partitive
K-Means
Self Organizing Map
Association Rules
Reinforcement Learning

Spark MLlib

Spark MLlib

Spark's scalable machine learning library consisting of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as underlying optimization primitives.

Spark MLlib: Highlights

- Summary statistics and correlation
- Hypothesis testing, random data generation
- Linear models of regression (SVMs, logistic and linear regression)
- Naive Bayes and Decision Tree classifiers
- Collaborative Filtering with ALS
- K-Means clustering
- SVD (singular value decomposition) and PCA
- Stochastic gradient descent

Summary of Spark Regression Models

- Linear regression
- Generalized linear regression
- Decision tree regression
- Random forest regression
- Gradient-boosted tree regression
- Survival regression
- Isotonic regression

Hands-On Lab

Tasks

- Linear regression
- Generalized linear regression
- Decision tree regression
- Random forest regression
- Gradient-boosted tree regression
- Survival regression
- Isotonic regression