Regression: Predict Continuous Data

Predict how a dependent variable (output, *t*) changes when any of the independent variables (inputs, or features, *x*) change, for example, how house prices change as a function of neighborhood and size, or how time spent on a web page varies as a function of the number of ads and content type. Training data has *N* samples and *D* features.
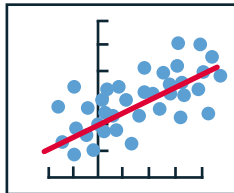
Linear Model $O(ND^2)$

Solves problems of the form:
$$y = w_0 + w_1 x_1 + \ldots + w_d x_d$$
with predicted value *y*, features, **x**, and fitted weights **w**.
Solved by minimizing "least square error", E_D:

$$E_D(\mathbf{w}, \mathbf{x}) = \sum_{n=0}^{N-1} (t_n - y_n)^2$$

On fitted models, access **w** as **model.coef_** and w_0 as **model.intercept_**.
Gotchas: Features must be uncorrelated, use **decomposition.PCA()**.
Code: **linear_model.LinearRegression()** if less than 100k samples, or see SGD.

Ridge $O(ND^2)$

When to use it: Less than 100k samples, noisy outputs.
How it works: Linear model that limits the size of the weights. Prevents overfitting by increasing bias. Minimizes *E* instead of E_D, where second term is called "L2 norm":

$$E(\mathbf{w}, \mathbf{x}) = E_D(\mathbf{w}, \mathbf{x}) + \alpha \frac{1}{2} \sum_{d=0}^{D-1} w_d^2$$

Code: **linear_model.Ridge(alpha)**
· **alpha**: Regularization strength, **alpha** > 0, corresponds to 1/*C* in other models. Increase if noisy samples.

Lasso $O(ND^2)$

When to use it: Less than 100k samples, only some features should be important.
How it works: Linear model that forces small weights to be zero. Minimizes *E* instead of E_D, where the second term is called "L1 norm":

$$E(\mathbf{w}, \mathbf{x}) = E_D(\mathbf{w}, \mathbf{x}) + \alpha \frac{1}{2} \sum_{d=1}^{D-1} |w_d|$$

Code: **linear_model.Lasso(alpha)**
· **alpha**: Regularization strength, **alpha** > 0, corresponds to 1/*C* in other models. Increase if noisy samples.
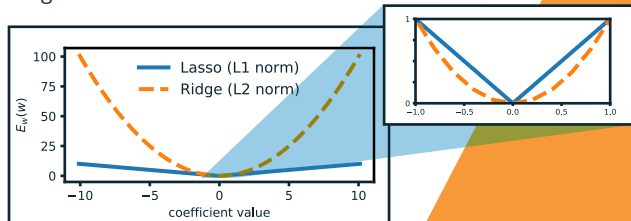**Tip:** Use with **feature_selection.SelectFromModel** as a transformation stage to select features with non-zero weights.

Ridge vs. Lasso – Shape of E_w

With Ridge and Lasso, the error to minimize *E* has an extra component E_w:

$$E(\mathbf{w}, \mathbf{x}) = E_D(\mathbf{w}, \mathbf{x}) + \alpha E_W(\mathbf{w})$$

Lasso produces sparse models because small weights are forced to zero.

Nonlinear Transformations

When to use them: "Straight line" not sufficient, for example, predicting temperature has a function of time of day.
How it works: "Reword" a nonlinear model in linear terms using *nonlinear basis functions*, $\Phi_j(x)$, so we can use linear model machinery to solve nonlinear problems. Linear model becomes:

$$y(\mathbf{w}, \mathbf{x}) = w_0 \phi_0(\mathbf{x}) + w_1 \phi_1(\mathbf{x}) + \ldots + w_j \phi_j(\mathbf{x})$$

Polynomial Expansion of Order P: E.g. A 2nd order polynomial two-feature model: $y(\mathbf{w}, \mathbf{x}) = (w_0 + w_1 x_1 + w_2 x_2)^2$
Becomes a model with these 6 basis functions:
$$\phi_0(\mathbf{x}) = 1, \ \phi_1(\mathbf{x}) = x_1, \ \phi_2(\mathbf{x}) = x_2, \ \phi_3(\mathbf{x}) = x_1^2, \ \phi_4(\mathbf{x}) = x_1 x_2, \ \phi_5(\mathbf{x}) = x_2^2,$$

Gotchas: Same feature affects many different coefficients, so an outlier can have a big global effect. Number of basis functions grows very quickly, $O((P+1)^{(D+1)})$.

Code: **poly=preprocessing.PolynomialFeatures(degree)**
x_poly = poly.fit_transform(x)

Radial Basis Functions (RBF): Local, Gaussian-shaped functions, defined by centers and width. Turns one feature into *P* features.

Code: **metrics.pairwise.rbf_kernel(x, centers, gamma)**
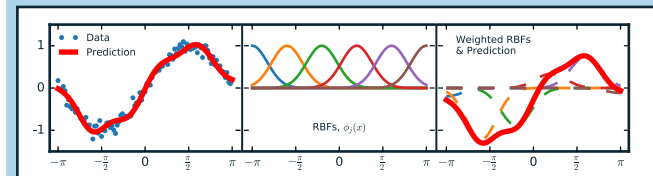
Support Vector Regressor $\sim O(N^2 D)$

When to use it: Many important features, more features than samples, nonlinear problem.
**How it works:** Find a function such that training points fit within a "tube" of acceptable error, with some tolerance towards points that are outside the tube.
Gotchas: Must scale inputs, see **StandardScalar** and **RobustScalar**.
Code: Start with **svm.LinearSVR(epsilon, C=1)**.
Make *C* smaller if lots of noisy observations (*C* = 1/*α*, small *C* means more regularization).
If LinearSVR doesn't work, use **svm.SVR(kernel='rbf', gamma)**.
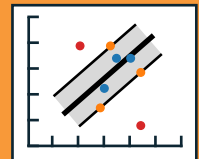
Stochastic Gradient Descent (SGD) Regressor

When to use it: Fit is too slow with other estimators.
How it works: "Online" method, learns the weights in batches, with a subset of the data each time. Pair with manual basis function expansion to train nonlinear models on really large datasets.
Code: **linear_model.SGDRegressor()** and **partial_fit()** method.
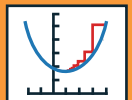
Performance Metrics in **sklearn.metrics**

mean_squared_error: Smaller is better. Puts large weight on outliers.
$$\frac{1}{N} \sum_{n=0}^{N-1} (t_n - y_n)^2$$

r2_score: Coefficient of determination. Best score is 1.0. Proportion of explained variance. Default for **model.score(x, t)**.
$$1 - \frac{\sum_{n=0}^{N-1} (t_n - y_n)^2}{\sum_{n=0}^{N-1} (t_n - \text{mean}(\mathbf{t}))^2}$$

mean_absolute_error: Smaller is better. Uses same scale as the data.
$$\frac{1}{N} \sum_{n=0}^{N-1} |t_n - y_n|$$

median_absolute_error: Robust to outliers.
$$\text{median}(|t_0 - y_0|, \ldots, |t_n, y_n|)$$

ENTHOUGHT

Classification: Predict Categorical Data

Predict the class, or label (*t*), of a sample based on its features (*x*). Examples: Recognize hand-written digits, or mark email as spam. In scikit-learn, labels are represented as integers and get expanded internally into matrices of binary choices between unique integer labels. Use `class_weight='balanced'` in most models to adjust for unbalanced datasets (more training data from one class than others). Training data has *N* samples and *D* features.

Logistic Regression $O(ND^2)$

When to use it: Need to understand contributions of features. Fast to train, easy to interpret.
**How it works:** Fits an s-shaped function (logistic function), which is continuous but has a steep transition between the two classes, and assigns class based on sign.
Gotchas: Inputs must be scaled and uncorrelated.
Code: `linear_model.LogisticRegression(C, solver)`.
- `penalty='l1'` to use estimator for feature selection.
- `solver='liblinear'` for small datasets or L1 penalty, `'lbfgs'`, `'sag'` or `'newton-cg'` for multi-class problems and large datasets, and `'sag'` for very large datasets.
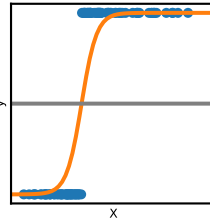
Decision Tree $O(ND\log(N))$

When to use it: Need to understand prediction decisions. Data has both continuous and categorical features. No scaling needed.
How it works: Chain binary decisions on increasingly smaller subsets of data. Deeper trees have more complex decision rules and a better fit.
Gotchas: Very often overfits. Consider doing dimensionality reduction beforehand. *N* must double with each extra level.
Code: `tree.DecisionTreeClassifier(max_depth)`. Start with `max_depth=3`, then increase. Use `tree.export_graphviz` to visualize tree.

| Yes | **known number?** | No |

is grandma?

pick up — ignore — ignore
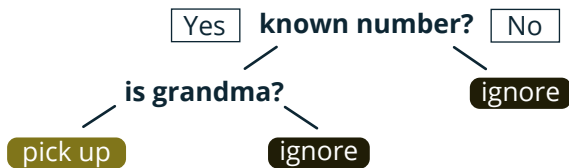
Ensemble Methods

When to use them: No single estimator gave satisfying results.
How they work: "Wisdom of the crowd". Combines predictions of multiple weak, biased estimators to create a better one. Two types: *averaging methods* build many estimators and average predictions; in *boosting methods* each new estimator tries to improve the previous one.
Gotchas: Hard to generate the perfect mix of estimators.
Code: All in **ensemble** module.

Averaging estimators:
- `RandomForestClassifier(max_features)`
- `ExtraTreesClassifier(max_features)`
Start with these, but always cross-validate:
- `max_features=sqrt(n_features)`
- `max_depth=None`
- `min_samples_split=1`

Boosting estimator:
- `AdaBoostClassifier()`
- `GradientBoostingClassifier()`

All:
- `Parallelize with n_jobs=-1`
  Increasing `n_estimators` is better, but slower

Support Vector Classifier $O(ND^2)$ to $O(ND^3)$

When to use it: Large number of features. Slightly more features than samples.
How it works: Maximize distance between classes in high-dimensional space, i.e., "maximum margin classifier".
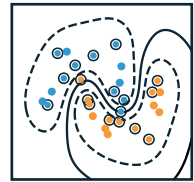**Gotchas:** Scale your data.
Code: `svm.SVC(kernel, C=1)`.
Make C smaller if lots of noisy samples.
If accuracy is important set `kernel='rbf'`.
If fast training is important, use `svm.LinearSVC()`.

Neighbor Classifiers $O(D\log N)$ to $O(DN)$

When to use them: Large datasets. Very irregular decision boundary.
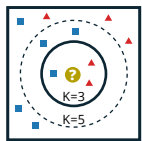**How it works:** Predict class by majority vote from nearby data.
Gotchas: Efficiency comes at the cost of also having high variance.
Code: `neighbors.KNeighborsClassifier(n_neighbors)`.
- Use `RadiusNeighborsClassifier()` for unbalanced data and *D* not too large.
- Try `weights='uniform'` and `'distance'`.

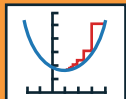## Stochastic Gradient Descent (SGD) Classifier

When to use it: Very large *N* and *D*, e.g., 10^5 samples and 10^5 features.
How it works: "Online" method, learns the weights in batches.
Gotchas: Data must be scaled.
Code: `linear_model.SGDClassifier(loss, alpha, n_iter)` and `partial_fit()` method.
- Use `n_iter=np.ceil(10**6/n_samples)`. `loss='hinge'` gives SVC, `'log'` gives logistic regression.

Performance Metrics in `sklearn.metrics`

They take targets, **t**, and predicted classes, **y**, as arguments. There's more than one way to be wrong. A fire alarm that always goes off is annoying, one that never goes off is costly.
- `confusion_matrix`: Explore how model confuses classes. Visualize with `seaborn.heatmap`.
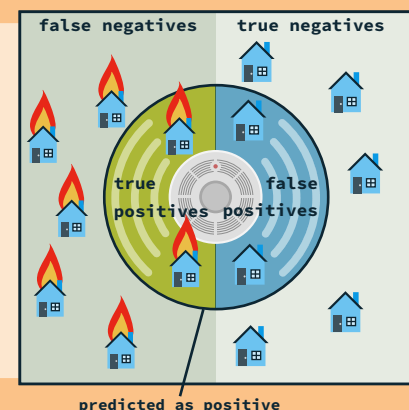
	7	0	3
	0	8	1
	3	1	6
	C1	C2	C3

- `accuracy_score` (default for `model.score`): Fraction correctly predicted. Meaningless if samples are unbalanced. (TP + TN) / Total

- `recall_score`: Fraction of predicted fire when there's actually fire. TP / (TP + FN)

- `precision_score`: Fraction of *correctly* predicted fire of all cases where fire is predicted. P predicted as P. TP / (TP + FP)

false negatives — true negatives — true positives — false positives — predicted as positive

ENTHOUGHT

Clustering: Unsupervised Learning

Predict the underlying structure in features, without the use of targets or labels. Splits samples into groups called "clusters". With no targets, models are trained by minimizing some definition of "distance" within a cluster. Data has N samples, D features, and the model discovers k clusters. Models can be used for prediction or for transformation, by reducing D features into one with k unique values.

Some models expect geometries that are "flat", or roughly spherical. Clusters with complicated shapes like rings or lines are not flat, and will not work in those models.
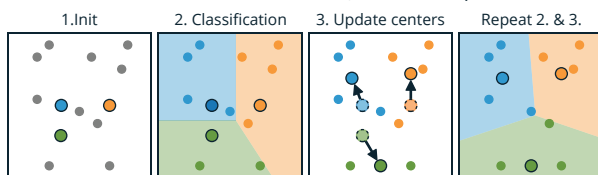

Flat   Non-Flat   Non-Flat
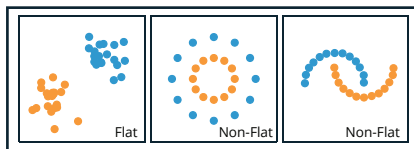
K-Means $O(kN)$

When to use it: Scales well. Works best on a small number of flat clusters. For large sample sizes, substitute MiniBatchKMeans.
How it works: Assigns samples to nearest of k cluster centers, then moves the centers to minize the average distance between centers and samples.
Gotchas: The K-Means algorithm used by scikit-learn is sensitive to initial location of the centers. Performs poorly on complex, non-flat shapes.
Code: `cluster.KMeans(n_clusters)`. `n_jobs=-1` to parallelize.

1. Init   2. Classification   3. Update centers   Repeat 2. & 3.

Mean Shift $O(N\log N)$

When to use it: Non-flat geometries. Unknown number of clusters. Need to guarantee convergence.
How it works: Finds local maxima given a window size.
Gotchas: Accuracy strongly tied to selecting correct window.
Code: `cluster.MeanShift(bandwith)`. Set bandwidth manually to small value for large dataset. Estimating it is $O(N^2)$ and can be the bottleneck.

Affinity Propagation $O(N^2)$

When to use it: Unknown number of clusters. Need to specify own similarity metric (**affinity** argument).
How it works: Finds data points which maximize similarity within cluster while minimizing similarity with data outside of cluster.
Gotchas: $O(N^2)$ memory use. Accuracy tied to **damping**.
Code: `cluster.AffinityPropagation(preference, damping)`
• **preference**: Negative. Controls the number of clusters. Explore on log scale.
• **damping**: 0.5 to 1.

DBSCAN $O(N^2)$

**When to use it:** Very non-flat geometries. Very uneven clusters.
How it works: Clusters are contiguous areas with high data density. Bounds of clusters are found using graph connectivity.
Gotchas: $O(N^2)$ memory use.
Not deterministic at cluster boundaries.
Code: `cluster.DBSCAN(min_samples, eps, metric)`
• Higher **min_samples**, or lower **eps** requires higher density to form a cluster.

Outlier   Core Samples
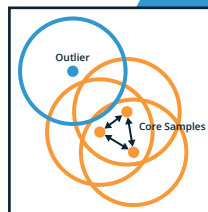
Agglomerative Clustering $O(N^2 \log N)$

When to use it: Need a flexible definition of distance (e.g. Levenshtein).
How it works: Defines all observations as unique clusters, then merges the closest ones iteratively.
**Gotchas:** Worst time complexity. "Rich get richer" behavior.
Code: `cluster.AgglomerativeClustering(linkage, affinity, connectivity)`. Set **linkage** criteria for merging:
• **'ward'**: minimize sum of square differences. Minimizes variance. Gives most regular cluster size.
• **'complete'**: minimize max distance between sample pairs.
• **'average'**: minimize average distance between all sample pairs. Yields uneven cluster sizes.
affinity: defines type of distances. **'l1'** for sparse features, e.g., text; **'cosine'** is invariant to scaling.
connectivity: provides extra constraints about which nodes can be merged, e.g., `neighbors.kneighbors_graph`.
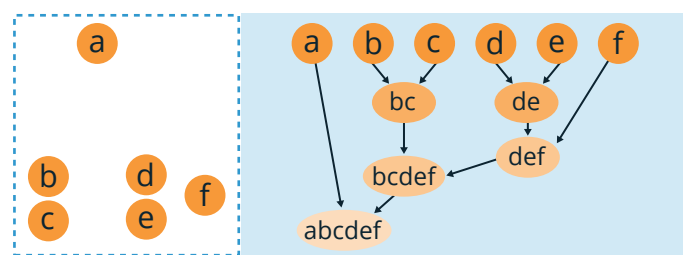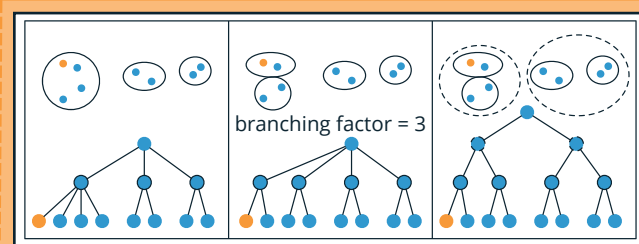

BIRCH $O(kN)$

When to use it: Large number of observations and small number of features.
How it works: Builds a balanced tree of groups of data, then clusters those groups instead of the raw data.
Gotchas: Performs poorly with large number of features.
Code: `cluster.Birch(threshold, branching_factor, n_clusters)`

branching factor = 3

Performance Metrics in `sklearn.metrics`

The metrics do not take into account the exact class values, only their separation. Score is based on *ground truth*, (targets), if available, or to a measure of similarity within class, and difference across classes.
Needs ground truth:
• **adjusted_rand_score**: -1 to 1 (best). 0 is random classes. Measures similarity. Related to accuracy (% correct).
• **adjusted_mutual_info_score**: 0 to 1 (best). 0 is random classes. 10x slower than **adjusted_rand_score**. Measures agreement.
• **homogeneity_completeness_v_measure**: 0 to 1 (best). **homogeneity**: each cluster only contains members of one class; **completeness**: all members of a class are in the same cluster: and, **v_measure_score**: the harmonic mean of both. Not normalized for random labeling. Doesn't need ground truth:
• **silhouette_score**: -1 to 1 (best). 0 means overlapping clusters. Based on distance to samples in same cluster and distance to next nearest cluster.

ENTHOUGHT