PATTERN RECOGNITION AND MACHINE LEARNING

Slide Set 7: Error Assessment and Regularization

February 2017

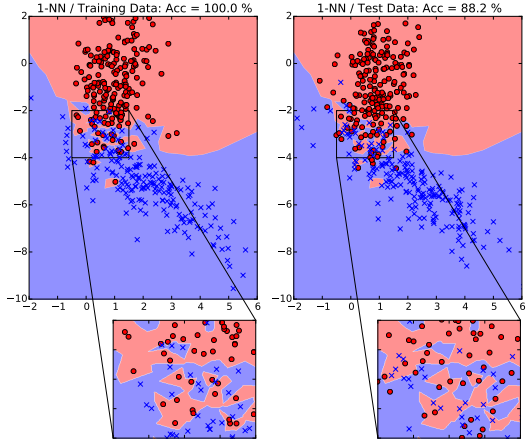
Heikki Huttunen

heikki.huttunen@tut.fi

Department of Signal Processing
Tampere University of Technology

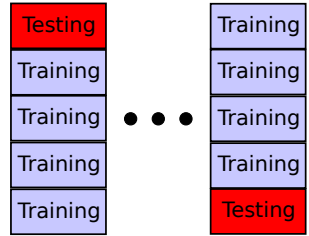
Generalization

- The important thing is how well the classifier works with unseen data.
- An overfitted classifier memorizes the training data and does not generalize.



Cross-validation

- The generalization ability of a classifier needs to be tested with **unseen** samples.
- In practice this can be done by splitting the data into separate training and test sets.
- A standard approach is to use *K*-fold cross-validation:
  - Split the training data to *K* parts (called folds)
  - Use each fold for testing exactly once and train with the other folds
  - The error estimate is the mean of the *K* estimates
- A common value for *K* is 5 or 10.



Cross-validation in sklearn

- Cross-validation is extremely elegant in sklearn.
- Essentially 1 line of code is required:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(clf, X, y)
```

- The cross-validator receives the classifier and the data and returns a list of scores for each test fold.
- The function can receive additional parameters that modify the number of folds, possible stratification (balancing the classes in folds), etc.

Code:

```
77 # Cross-validate to estimate accuracy
78
79 scores = cross_val_score(clf,
80                          X,
81                          y,
82                          cv = 10,
83                          n_jobs = 2)
84
85 for fold in range(10):
86     print "Fold %d score: %.2f %%" % \
87           (fold, 100*scores[fold])
88
89 print "-" * 10
90 print "CV accuracy: %.2f %%" % \
91       (100*np.mean(scores))
```

Result:

```
Fold 0 score: 95.65 %
Fold 1 score: 100.00 %
Fold 2 score: 100.00 %
Fold 3 score: 100.00 %
Fold 4 score: 86.36 %
Fold 5 score: 100.00 %
Fold 6 score: 100.00 %
Fold 7 score: 100.00 %
Fold 8 score: 100.00 %
Fold 9 score: 90.48 %
----------
CV accuracy: 97.25 %
```

Stratified Cross-validation

- The CV split can be controlled using the cv argument.
- Instead of a completely random split, a **stratified** split is preferred: Each fold will hold a mixture of classes in same proportion as in full dataset.
- The stratified split can be generated easily:

```
>>> from sklearn.model_selection import StratifiedKFold
# Generate SKF split. This needs the class labels y.
>>> skf = StratifiedKFold(y, 10)

# skf is a generator; we can transform it into a list.
>>> print list(skf)[0]
(array([0, 1, 4, ... , 399]), array([2, 3, 10, ... , 390]))
# Contains 10 train-test index pairs, above the first one.
# Each index set has same proportion of samples from each class.
```

```
# CV score estimation
from sklearn.model_selection import
     cross_val_score, StratifiedKFold

skf = StratifiedKFold(y, 10, shuffle = True)
scores = cross_val_score(clf, X, y,
                         cv = skf)
```

```
# Print scores

>>> print ("Accuracy: %.2f +- %.2f" %
          (np.mean(scores),
           np.std(scores)))

Accuracy: 0.91 +- 0.04
```

Leave-One-Out CV

- A popular accuracy estimation strategy is to use **Leave-One-Out** (LOO) split.
- Extreme case of *K* fold CV with *K* equal to number of samples.
- Requires lot of computation: If there are *N* samples in training set, we need to train *N* times.
- Also: surprisingly large variance.

```
# CV score estimation
from sklearn.model_selection import LeaveOneOut

loo = LeaveOneOut(y.size)
scores = cross_val_score(clf, X, y,
                         cv = loo)
```

```
# Print scores

>>> print ("Accuracy: %.2f +- %.2f" %
          (np.mean(scores),
           np.std(scores)))

Accuracy: 0.92 +- 0.28
```

Leave-One-Group-Out CV

- Sometimes the data is recorded from multiple sources, and we wish to test the generalization over groups.
- For example: Data is recorded in 10 sessions; we wish to train with 9 and test with one.
- Or: we get images from 10 cameras and would like to have a model that is robust when the 11th camera is added.

```
# CV score estimation
from sklearn.model_selection import
       LeaveOneGroupOut

# Suppose these are these (e.g., camera indices
       per sample).
labels = [1,1,1,2,2,2,3,3,3,3,4,4]
logo = LeaveOneGroupOut(labels)
scores = cross_val_score(clf, X, y, cv = logo)
```

Other Error Metrics

- Error estimation is not limited to accuracy.
- Any error metric can be used instead.
- Popular ones are invoked easily, but you can also implement your own.
  - `'accuracy'`: Accuracy score (default)
  - `'roc_auc'`: Area under ROC curve
  - `'recall'`: How many % of positive samples were found
  - `'precision'`: How many % of found samples were positive
  - `'f1'`: F_1 score; harmonic mean of recall and precision

```
# Code:

metrics = ['accuracy',
           'roc_auc',
           'recall',
           'precision',
           'f1']

for scorer in metrics:
    scores = cross_val_score(clf,
                             X,
                             y,
                             scoring = scorer)

    print ("%s score: %.2f +- %.2f" % \
          (scorer,
           np.mean(scores),
           np.std(scores)))
```

```
# Result:

accuracy score: 0.92 +- 0.01
roc_auc score: 0.98 +- 0.00
recall score: 0.90 +- 0.03
precision score: 0.93 +- 0.01
f1 score: 0.91 +- 0.02
```

Cross-validating a set of Classifiers

- A nice property of sklearn is that each predictor conforms to the same interface (*i.e.*, each implements `.fit()`, and `.predict()` methods).
- Thus, we can examine a list of them in a `for` loop easily.

Code:

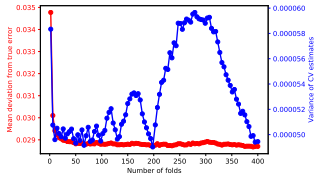
```
# Test a collection of classifiers

classifiers = [LogisticRegression(),
               LDA(),
               LinearSVC(),
               SVC(),
               RandomForestClassifier(),
               KNeighborsClassifier()
               ]

names = ['LogReg',
         'LDA',
         'LinSVM',
         'SVM',
         'RF',
         'NN']

for i, clf in enumerate(classifiers):

    scores = cross_val_score(clf,
                             X,
                             y,
                             cv = 10,
                             n_jobs = 2)
    print "Classifier %s: score = %.2f %%" % \
          (names[i], 100*np.mean(scores))
```

Result:

```
Classifier LogReg: score = 97.25 %
Classifier LDA: score = 94.39 %
Classifier LinSVM: score = 97.25 %
Classifier SVM: score = 79.82 %
Classifier RF: score = 89.35 %
Classifier NN: score = 88.50 %
```

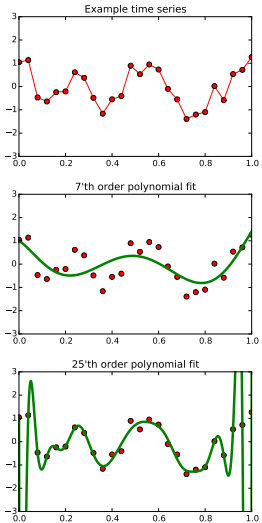
Selection of K in cross-validation

- There is no definitive answer to the choice of number of folds in CV.
- The best K depends on the amount of data, difficulty of the problem, error metric, etc.
- In general, more folds is better, although the variance starts to increase when approaching leave-one-out.
- Attached is a figure of difference between CV error and true error (which we happen to know in our artificial data case).



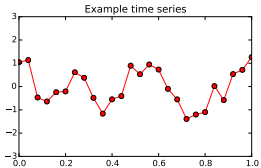
Overfitting

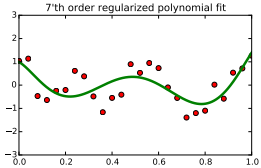
- Generalization is also related to *overfitting*.
- On the right, a polynomial model is fitted to a time series to minimize the error between the samples (red) and the model (green).
- As the order of the polynomial increases, the model starts to follow the data very faithfully.
- Low-order models do not have enough expression power.
- High-order models are over-fitting to noise and become "unstable" with crazy values near the boundaries.

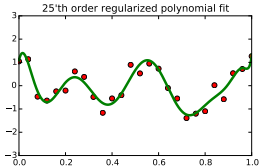


Regularization

- Regularization adds a penalty term to the fitting error.
- The model is encouraged to use small coefficients.
- Large coefficients are expensive, so the model can afford to fit only to the major trends.
- On the right, the high order model has a good expression power, but does still not follow the noise patterns.







Regularization in Regression

- Regularization techniques were first assessed in a regression context with linear models.
- Linear regression model assumes that the inputs $\mathbf{x}_n \in \mathbb{R}^P$ and outputs $\mathbf{y}_n \in \mathbb{R}$ are related as

$$y_n = \mathbf{w}^T \mathbf{x}_n + e_n,$$

  where $\mathbf{e}_n \sim \mathcal{N}(0, \sigma^2)$ is the error term.
- With these assumptions, *least squares* returns the optimal solution:

$$\underset{\mathbf{w}}{\text{minimize}} \left(\sum_{n=0}^{N-1} (y_n - \mathbf{w}^T \mathbf{x}_n)^2 \right)$$

- There exists a closed form solution for LS:

$$\mathbf{w}_{\text{LS}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}.$$

with $\mathbf{X} = \left[\mathbf{x}_0^T, \mathbf{x}_1^T, \ldots, \mathbf{x}_{N-1}^T \right]^T$.

Regularization in Regression

- The most straightforward regularization technique adds a squared penalty with weight $\lambda > 0$ for the weights:

$$\underset{\mathbf{w}}{\text{minimize}} \left(\sum_{n=0}^{N-1} (y_n - \mathbf{w}^T \mathbf{x}_n)^2 + \lambda \mathbf{w}^T \mathbf{w} \right).$$

- This is called *Tikhonov regularization* or *Ridge Regression*, and there is a closed form solution:

$$\mathbf{w}_{\text{RIDGE}} = (\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y}.$$

- Another name is L_2 regularization, because we take the L_2 norm of the weights.
- In general, the L_q norm (for $q > 0$) is defined as

$$\|\mathbf{w}\|_q = \left(|w_0|^q + |w_1|^q + \cdots + |w_{P-1}|^q \right)^{-q}$$

- See `sklearn.linear_model.Ridge`.

L1 Regularization in Regression

- More recently, L_1 penalty has become popular—simply change the squared penalty into absolute penalty:

$$\underset{\mathbf{w}}{\text{minimize}} \left(\sum_{n=0}^{N-1} (y_n - \mathbf{w}^T \mathbf{x}_n)^2 + \lambda \|\mathbf{w}\|_1 \right),$$

  with $\|\mathbf{w}\|_1 = \sum_p |w_p|$.
- This is called *LASSO penalty*[1].
- With this penalty, there is no closed form expression.
- The minimum is solved iteratively using, *e.g.,* gradient search.
- See `sklearn.linear_model.Lasso`.

[1] (Least Absolute Shrinkage and Selection Operator)

L_2 Regularization with Linear Classifiers

- The same regularizers apply also for linear classifiers.
- L_2 **penalized Logistic Regression**:

$$\text{penalized log-loss} = \sum_{n=0}^{N-1} \underbrace{\ln(1+\exp(y_n \mathbf{w}^T \mathbf{x}_n))}_{\text{log-loss}} + \lambda \underbrace{\mathbf{w}^T\mathbf{w}}_{\text{penalty}}$$

- L_2 **penalized Linear SVM**:

$$\text{penalized hinge loss} = \sum_{n=0}^{N-1} \underbrace{\max(0, 1-y_n \mathbf{w}^T \mathbf{x}_n)}_{\text{hinge loss}} + \lambda \underbrace{\mathbf{w}^T\mathbf{w}}_{\text{penalty}}$$

- The coefficient $\lambda \geq 0$ balances the strength of regularization:
  - Large λ: small coefficients (heavy penalty for large $\mathbf{w}^T\mathbf{w}$)
  - Small λ: large coefficients (small penalty for large $\mathbf{w}^T\mathbf{w}$)

L_1 Regularization with Linear Classifiers

- More recently, research has concentrated around the L_1 norm: $\|\mathbf{w}\|_1 = \sum |w_k|$.
- L_1 **Regularized Logistic Regression**:

$$\text{penalized log-loss} = \sum_{n=0}^{N-1} \underbrace{\ln(1+\exp(y_n \mathbf{w}^T \mathbf{x}_n))}_{\text{log-loss}} + \lambda \underbrace{\|\mathbf{w}\|_1}_{L_1 \text{ penalty}}$$

- L_1 **Regularized Linear SVM**:

$$\text{penalized log-loss} = \sum_{n=0}^{N-1} \underbrace{\max(0, 1-y_n \mathbf{w}^T \mathbf{x}_n)}_{\text{hinge loss}} + \lambda \underbrace{\|\mathbf{w}\|_1}_{L_1 \text{ penalty}}$$

Sparsity

- A key benefit of L_1 is that it promotes *sparse* weight vectors; *i.e.,* most entries of $\mathbf{w}$ are zero.
- This is equivalent to **feature selection:** most features are multiplied by zero, thus discarding them completely.
- Feature selection property can be understood from an alternative formulation of the optimization problem.
- The following two problems are equivalent

$$\text{minimize} \sum_{n=0}^{N-1} \ell(\mathbf{w}) + \lambda \|\mathbf{w}\|_p$$

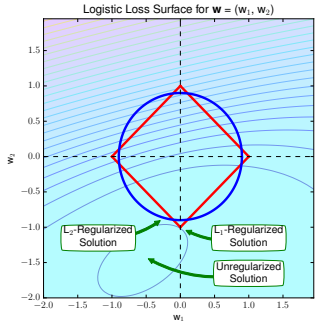
and

$$\text{minimize} \sum_{n=0}^{N-1} \ell(\mathbf{w}) \text{ such that } \|\mathbf{w}\|_p \leq C$$

with $\ell(\mathbf{w})$ either the log loss or hinge loss and $p \in \{1, 2\}$.

Sparsity

- The parameters λ and C are obviously different.
- However, there is a one-to-one mapping: for each λ we can find the corresponding C that gives the same solution.
- In the latter formulation, we try to find the weight vector that minimizes the loss inside the region $\|\mathbf{w}\|_p \leq C$.
- This region is circular for L_2 and diamond shaped for L_1.
- With L_1, the solutions are often at the corners of the diamond.

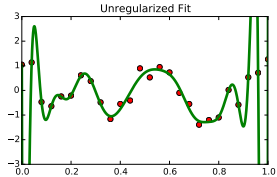


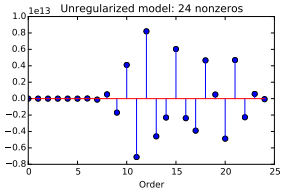
The L_1 solution tends to lie along the coordinate axes, where some of the coefficients are zero.

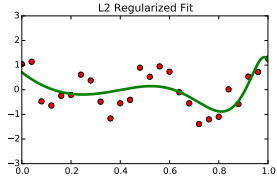
This is more likely if the area size decreases (or $C \to 0$).

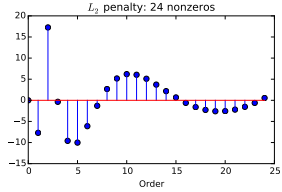
Sparsity

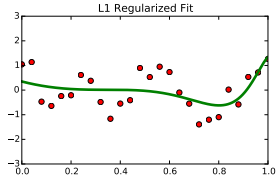
- The plots illustrate the model coefficients without regularization, with traditional regularization and sparse regularization.
- The importance of sparsity is twofold: The model can be used for feature selection, but often also generalizes better.

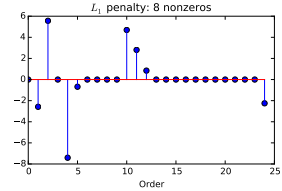






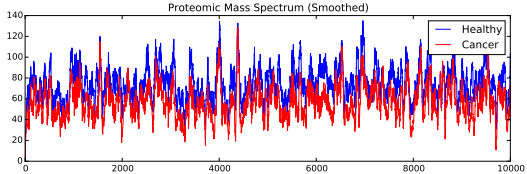






Regularization in Ovarian Cancer Detection

- We will study an example of classifying proteomic fingerprints of mass-spectra measured from ovarian cancer patients and healthy controls.[2]
- 121 cancer patients and 95 healthy controls.
- Raw mass spectra consists of 10000 measurements.



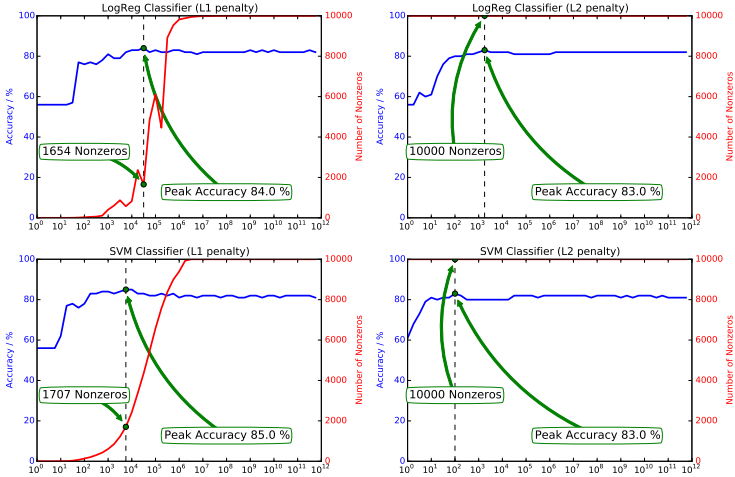
[2] Conrads, Thomas P., et al. "High-resolution serum proteomic features for ovarian cancer detection." *Endocrine-Related Cancer* (2004).

Training a Classifier

```
classifiers = [(LogisticRegression(), "LogReg"),
               (LinearSVC(dual = False), "SVM")]
C_range = 10.0 ** np.arange(0,12, 0.25)

for clf, name in classifiers:

    for penalty in ["l1", "l2"]:
        clf.penalty = penalty

        accuracies = []
        nonzeros = []

        for C in C_range:

            clf.C = C
            clf.fit(X_train, y_train)
            prediction = clf.predict(X_test)
            accuracy = 100.0 * np.mean(prediction == y_test)
```

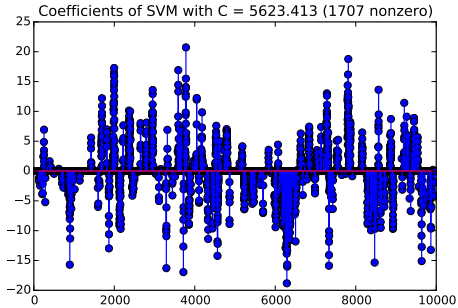
- The attached code trains and applies a LR and SVM classifiers for the ovarian cancer problem.
- We loop over two penalties and a range of regularization parameters *C*.

Results


LogReg Classifier (L1 penalty)
Accuracy / %
Number of Nonzeros
1654 Nonzeros
Peak Accuracy 84.0 %
LogReg Classifier (L2 penalty)
10000 Nonzeros
Peak Accuracy 83.0 %
SVM Classifier (L1 penalty)
1707 Nonzeros
Peak Accuracy 85.0 %
SVM Classifier (L2 penalty)
10000 Nonzeros
Peak Accuracy 83.0 %

Analysis of the Classifier

- The ℓ_1 regularized Linear SVM was the most accurate classifier.
- It is defined by its coefficients **w**.
- The nonzero coefficients correspond to the peaks of the mass spectra.



Feature Selection

- One of the benefits of L_1 regularization is its ability for *feature selection*.
- More specifically, L_1 can choose the most essential set of good features and discard the rest.
- Helps in high-dimensional cases.
- Improves performance by removing "confusers"; *i.e.,* measurements which have no predictive value (or may even degrade performance).

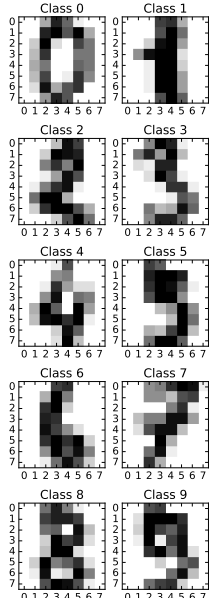
Traditional Approaches to Feature Selection

- **Variance based selection**: Retain features with high variance.
  - Simple to implement; however: high variance may not imply importance.
  - `sklearn.feature_selection.VarianceThreshold`
- **Statistics based selection**: Apply statistical tests for dependence between features and labels, *e.g.,* chi-squared test.
  - Good if the assumptions are correct (often not the case).
  - `sklearn.feature_selection.SelectKBest`.
- **Recursive selection**: Progressively add or remove features that seem to improve performance the most.
  - Forward selection starts with empty set and adds variables one by one.
  - Backward elimination starts with full set and removes variables one by one.
  - There are also hybrid versions that alternate between addition and removal.
  - `sklearn.feature_selection.RFECV` implements this (with cross-validation).

Example

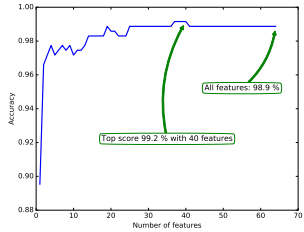
- Consider an example of classifying the *digits* dataset (`sklearn.datasets.load_digits`).
- We use LDA classifier with recursive feature elimination.
- For simplicity, we consider only two classes (zeros and ones).

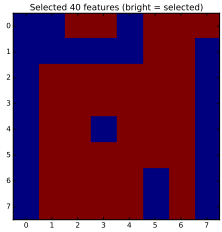
```
from sklearn.datasets import load_digits
from sklearn.lda import LDA
from sklearn.feature_selection import RFECV

digits = load_digits()

# Use only classes 8 and 9
X = digits.data[digits.target > 7, :]
y = digits.target[digits.target > 7]

# Select features
rfecv = RFECV(estimator=LDA())
rfecv.fit(X, y)

# Scores and feature sets are here
scores = rfecv.grid_scores_
mask = rfecv.support_.reshape(8, 8)
```



Results

- The smallest high scoring set of features consists of 43 pixels.
- There are also larger sets with equal score.
- However, using all features will give a lower score.

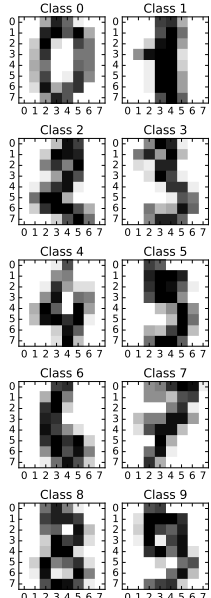




Example with L1-LR

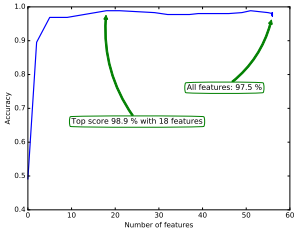
- Let's try to use L_1 regularized logistic regression for the same task.

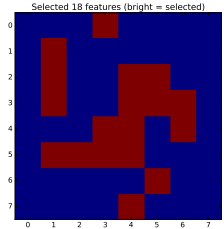
```
from sklearn.datasets import load_digits
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

digits = load_digits()

# Use only classes 8 and 9
X = digits.data[digits.target > 7, :]
y = digits.target[digits.target > 7]

# Select features
clf = LogisticRegression(penalty = "l1")
C_range = 10.0 ** np.arange(-5,6, 0.5)

accuracies = []
nonzeros = []
bestScore = 0

for C in C_range:
    clf.C = C
    score = cross_val_score(clf, X, y, cv = 5).mean()
```



Example

- The smallest high scoring set of features consists of 50 pixels.
- Note: a different set will result at each run due to randomness of CV-5. Also the below set is different from the CV results—trained with all data.
- The L_1 feature set seems ‘better’ than RFE.





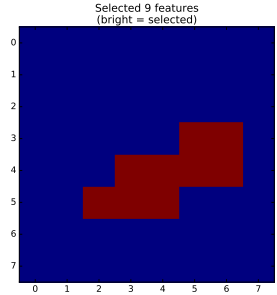
Randomized LogReg for Feature Selection

- Basic L_1 feature selection has been extended through randomization.
- The randomized version iterates by showing subsamples of the training set and observes which features get selected consistently.
- More robust than normal L_1; also called *stability selection*.
- See `sklearn.linear_model.RandomizedLogisticRegression` for classification.
- See `sklearn.linear_model.RandomizedLasso` for regression.

Example

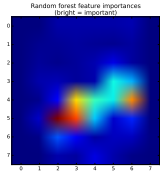
```
from sklearn.linear_model import RandomizedLogisticRegression

# Train the model 1000 times, always sampling 75% of all data
# After 1000 runs, the feature is chosen if it appears
# in at least 25% of the feature sets.
# C is the L1 regularization parameter for LogReg.

model = RandomizedLogisticRegression(C = 1,
                                     sample_fraction = 0.75,
                                     n_resampling = 1000,
                                     selection_threshold=0.25,
                                     n_jobs = 2,
                                     random_state = 42)

model.fit(X, y)
mask = model.get_support() # 64 binary variables.
mask = mask.reshape(8,8)
```

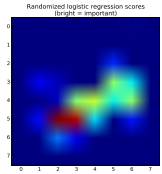
- The benefit of Randomized LogReg is its stability.
- The selected features are very similar on different test runs.

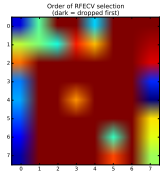


Feature Ranking

- Often, we are interested in the relative importance of features—not only which where selected, but in which order.
- Random forest derives this by shuffling each feature and studying the effect in accuracy: `RandomForestClassifier.feature_importances:`.
- Randomized Logistic Regression keeps track of how often each feature was selected: `RandomizedLogisticRegression.scores_`.
- Also, the RFECV keeps track of the order in which features were dropped: `RFECV.ranking_`.
- The plots for our data are shown on the right.





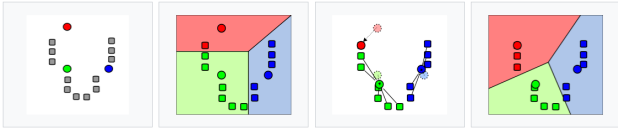


Unsupervised learning

- Unsupervised learning does not require class labels, but instead attempts to understand the data without them.
- Unsupervised learning has many uses, such as:
  1. Finding clusters within the data
  2. Transforming the data representation more suitable for later use; *e.g.*, dimensionality reduction
- Let’s study each of them in more detail.

Clustering

- K-means is a standard algorithm for splitting unlabeled data into clusters.
- K-means requires the number of assumed clusters as input.



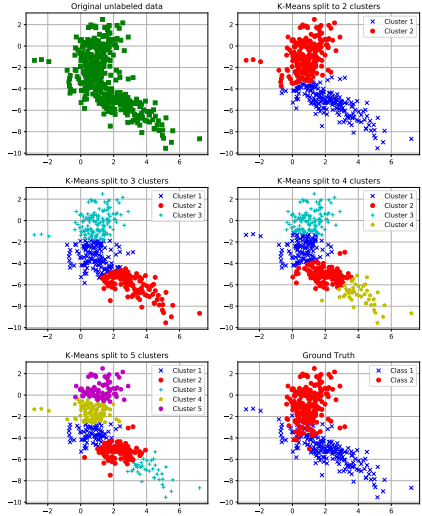
1. *k* initial "means" (in this case *k*=3) are randomly generated within the data domain (shown in color).

2. *k* clusters are created by associating every observation with the nearest mean. The partitions here represent the Voronoi diagram generated by the means.

3. The centroid of each of the *k* clusters becomes the new mean.

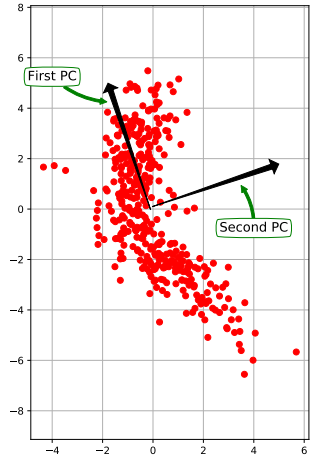
4. Steps 2 and 3 are repeated until convergence has been reached.

Figure credits: Wikipedia::K-means clustering



Dimensionality reduction

- High dimensional data may have a lot of redundancy (correlation) between features.
- This can be compressed by mapping to a lower dimension while preserving most of the variation.
- **Principal component analysis** (PCA) is a classical approach for this.
- PCA finds the directions, where the data has maximum variation.
- In the figure, the first principal component (PC) is the direction with highest variance.
- The second PC is the next (orthogonal direction) with second highest variance.



The PCA

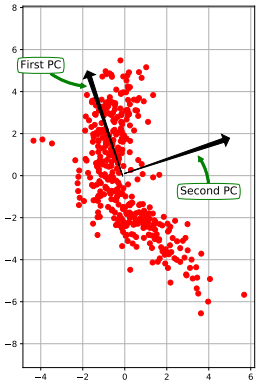
- It can be shown that the PCA axes are eigenvectors of covariance matrix:

$$\mathbf{V} = \text{eig}(\mathbf{X}^T\mathbf{X})$$

- In numpy terms, this is only one line of code.

```
from numpy.linalg import eig
from numpy import cov

# Assume our data is in X with shape [400, 2].
D, V = eig(cov(X, rowvar = False))

print("First PC: %s" % str(V[:, 0]))
print("Second PC: %s" % str(V[:, 1]))
```

```
First PC: [-0.33683673  0.94156307]
Second PC: [ 0.94156307  0.33683673]
```

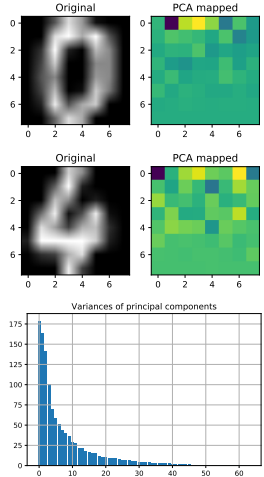


- Note: the PC's may not be in the order of decreasing variance. Use eigenvalues `D` to reorder: `V = V[:, np.argsort(D)]`. Then, `V[:, -1]` will be the 1st PC.

Understanding the PCA

- The matrix V of previous slide transforms the data to a new coordinate system.
- In other words, multiplication by V only rotates the coordinate axes.
- Below we fit a PCA into the 64-dimensional digits dataset.

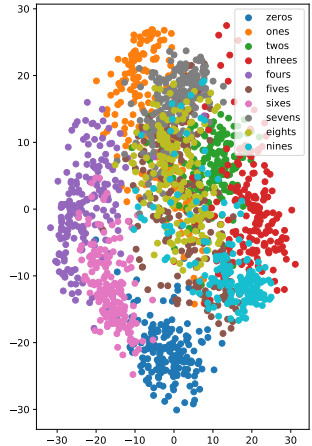
```
from sklearn.datasets import load_digits

digits = load_digits()
X = digits.data

# Compute 64x64 PCA matrix V:
D, V = eig(cov(X, rowvar = False))

# Subtract mean from each column:
X = X - np.tile(X.mean(axis = 0), [X.shape[0], 1])

# Map X through the PCA (rotate axes):
X_rot = np.matmul(X, V)
```



PCA in Dimensionality Reduction

- The first 2 principal components cover 28.5 % of all variance.
- So maybe they also contain most of the information of the bitmaps.
- Let’s see how the 2 components reside in the 2D space.

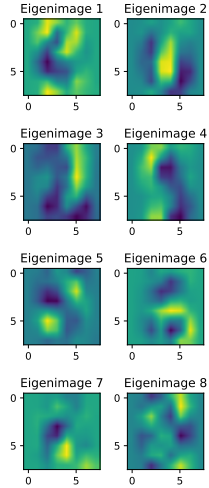
```
# Assume X_rot contains the PCA of previous slide.

# Retain only 2 components and plot
X_2D = X_rot[:, :2]

labels = ['zeros', 'ones', 'twos', 'threes', 'fours', 'fives', 'sixes', 'sevens', '
       eights', 'nines']

for k in range(10):
    plt.scatter(X_2D[y == k, 0], X_2D[y == k, 1], label = labels[k])

plt.legend(loc = 'best')
```



Eigenimages

- The first principal components represent templates, against which we correlate each frame.
- How do these *eigenimages* look like?

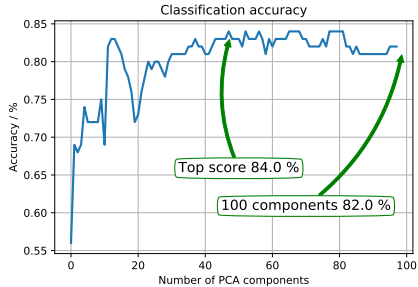
```
fig, ax = plt.subplots(4, 2, figsize = [3,7])

for k in range(4):
    for j in range(2):
        ax[k][j].imshow(V[:, k].reshape(8,8))
        ax[k][j].set_title("Eigenimage %d" % (k * 2 + j + 1))
```



PCA in Classification

- Let’s try to classify the arcene dataset with PCA dimensionality reduction.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.metrics import accuracy_score
from sklearn.decomposition import PCA

# Remove Mean
normalizer = Normalizer()
normalizer.fit(X_train)
X_train = normalizer.transform(X_train)
X_test  = normalizer.transform(X_test)

# Use sklearn PCA, because it is faster.
pca = PCA()
pca.fit(X_train)
X_train = pca.transform(X_train)
X_test  = pca.transform(X_test)

accuracies = []

for num_components in range(1, 100):
    clf = LinearDiscriminantAnalysis()
    clf.fit(X_train[:, :num_components], y_train)
    y_pred = clf.predict(X_test[:, :num_components])
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)
```



Hyperparameter Optimization

- The most common hyperparameter optimization techniques are:
  1. **Grid search:** Try each hyperparameter combination in a fixed grid. For example, try SVM with kernels ’linear’ and ’rbf’ and with C = [1,10,100,1000] (total 8 combinations).
  2. **Random search:** Randomize the hyperparameters. For example, select the kernel at random from {’linear’, ’rbf’} and C from the range [1, 1000]. Iterate *N* times.
  3. **Optimized grid search:** Instead of trying all combinations on a fixed grid, try to cover the search space as uniformly as possible.
  4. **Reactive search:** Use the results of previous iterations for choosing the next hyperparameter combination. For example, previous iterations suggest that ’rbf’ gives better scores than ’linear’, so it is favored in subsequent trials.

Grid Search

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

# Define two grids; for linear and rbf
grid1 = {'kernel': ['linear'], 'C': [1, 10, 100, 1000]}
grid2 = {'kernel': ['rbf'], 'C': [1, 10, 100, 1000],
         'gamma': [0.001, 0.0001]}
param_grid = [grid1, grid2]

clf = GridSearchCV(SVC(),
                   param_grid,
                   cv=5,
                   scoring = 'accuracy')
clf.fit(X, y)
```

```
# Scores for the digits data:
0.973 kernel = rbf / C = 10 / gamma = 0.001
0.973 kernel = rbf / C = 100 / gamma = 0.001
0.973 kernel = rbf / C = 1000 / gamma = 0.001
0.972 kernel = rbf / C = 1 / gamma = 0.001
0.963 kernel = rbf / C = 100 / gamma = 0.0001
0.963 kernel = rbf / C = 1000 / gamma = 0.0001
0.959 kernel = rbf / C = 10 / gamma = 0.0001
0.949 kernel = linear / C = 1
0.949 kernel = linear / C = 10
0.949 kernel = linear / C = 100
0.949 kernel = linear / C = 1000
0.948 kernel = rbf / C = 1 / gamma = 0.0001
```

- Although grid search is easy to implement with for loops, sklearn has a more structured approach.
- `GridSearchCV` allows to specify a number of grids and cross-validates automatically.

Random Search

```
import scipy
from sklearn.grid_search import RandomizedSearchCV

# Specify parameters and distributions to sample from
param_grid = {"C": scipy.stats.expon(loc=0, scale = 5),
              "kernel": ['linear', 'rbf'],
              "gamma": scipy.stats.expon(loc = 0, scale = 0.1)}

# Run randomized search 10 times
num_iters = 10

clf = RandomizedSearchCV(SVC(),
                         param_distributions=param_grid,
                         n_iter=num_iters)
clf.fit(X, y)
```

```
# Scores for the digits data:
0.94 kernel = linear / C = 8.270861 / gamma = 0.1191563
0.94 kernel = linear / C = 0.7353522 / gamma = 0.1815501
0.94 kernel = linear / C = 1.793605 / gamma = 0.03298398
0.94 kernel = linear / C = 4.12874 / gamma = 0.09421099
0.94 kernel = linear / C = 5.952222 / gamma = 0.0689759
0.71 kernel = rbf / C = 1.193375 / gamma = 0.009067043
0.11 kernel = rbf / C = 8.202399 / gamma = 0.05592028
0.10 kernel = rbf / C = 0.3224507 / gamma = 0.0861419
0.10 kernel = rbf / C = 0.9282524 / gamma = 0.07795433
0.10 kernel = rbf / C = 0.02467871 / gamma = 0.1224617
```

- `RandomizedSearchCV` uses the same interface as `GridSearchCV`.
- Although the result is not as good in this case, this becomes effective when training is slow (*e.g.*, training a deep net takes 3 days.
- Many papers suggest that random search is more effective than grid search (or even human guided search).

Advanced Approaches

- When one training iteration is slow, advanced approaches may become handy.
- However, you may end up optimizing the hyper-hyperparameters (parameters of the hyperparameter selection algorithm).
- **hyperopt** is a popular package for advanced parameter sampling.
- hyperopt implements currently two algorithms that try to cover the search space more efficiently than the plain grid search.
- **spearmint** implements a Bayesian optimization algorithm.
- Based on Gaussian process priors, trying to suggest better combinations using the previous ones.

APPLICATION EXAMPLES

Applications

- **Application example 1:** ICANN MEG Mind Reading Challenge, June 2011
- **Application example 2:** IEEE MLSP 2012 Birds competition, Sept. 2013
- **Application example 3:** DecMeg2014: Decoding the Human Brain, July 2014

ICANN MEG Mind Reading Challenge

- We participated in the Mind reading challenge from MEG data organized in the ICANN 2011 conference. [3] [4] [5]

Rank	Team	Affiliation	Accuracy (%)
1.	Huttunen et al.	Tampere University of Technology	68.0
2.	Santana et al.	Universidad Politecnica de Madrid	63.2
3.	Jylänki et al.	Aalto University	62.8
4.	Tu & Sun (1)	East China Normal University	62.2
5.	Lievonen & Hyötyniemi	Helsinki Institute for Information Technology	56.5
6.	Tu & Sun (2)	East China Normal University	54.2
7.	Olivetti & Melchiori	University of Trento	53.9
8.	Van Gerven & Farquhar	Radboud University Nijmegen	47.2
9.	Grozea	Fraunhofer Institute FIRST	44.3
10.	Nicolaou	University of Cyprus	24.2

[3] H. Huttunen *et al.*, "Regularized logistic regression for mind reading with parallel validation," *Proc. ICANN/PASCAL2 Challenge: MEG Mind-Reading. Aalto University publication series*, Espoo, June 2011.

[4] H. Huttunen *et al.*, "MEG Mind Reading: Strategies for Feature Selection," in *Proc. of The Federated Computer Science Event 2012*, Helsinki, May 2012.

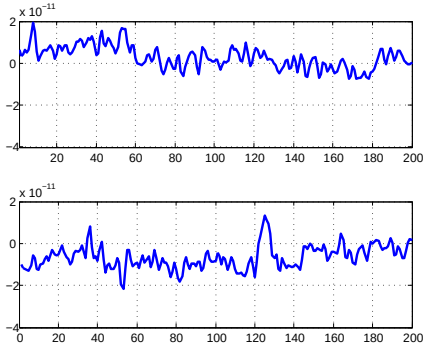
[5] H. Huttunen *et al.*, "Mind Reading with Regularized Multinomial Logistic Regression," *Machine Vision and Applications*, Aug. 2013

ICANN MEG Mind Reading Challenge Data

- The competition data consists of MEG recordings while watching five different movie categories.
- The data contains measurements of 204 planar gradiometer channels at 200Hz rate, segmented into samples of one second length.
- The task was to design and implement a classifier that takes as an input the MEG signals and produces the predicted class label.
- The training data with annotations had 727 one-second samples, and the secret test data 653 unlabeled samples.
- 50 samples of the training data were special, because they were recorded on the same day as the test data.

The Curse of Dimensionality

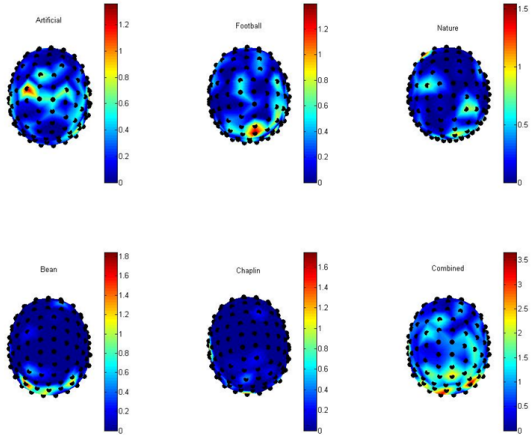
- Since each measurement is a time series, it cannot be fed to classifier directly.
- Instead, we calculated a number of common quantities.
- In all, our pool of features consists of $11 \times 204 = 2244$ features. This means $2244 \times 5 = 11220$ parameters.



Intercept	$x_i^{(1)} = \hat{b}_i$
Slope	$x_i^{(2)} = \hat{a}_i$
Variance (d)	$x_i^{(3)} = \frac{1}{N}\sum_{n=1}^{N} \tilde{s}_i^2(n)$
Std. dev. (d)	$x_i^{(4)} = \sqrt{x_i^{(3)}}$
Skewness (d)	$x_i^{(5)} = \frac{1}{N}(x_i^{(4)})^{-3}\sum_{n=1}^{N} \tilde{s}_i^3(n)$
Kurtosis (d)	$x_i^{(6)} = \frac{1}{N}(x_i^{(4)})^{-4}\sum_{n=1}^{N} \tilde{s}_i^4(n)$
Variance	$x_i^{(7)} = \frac{1}{N}\sum_{n=1}^{N}(s_i(n) - \hat{b}_i)^2$
Std. dev.	$x_i^{(8)} = \sqrt{x_i^{(7)}}$
Skewness	$x_i^{(9)} = \frac{1}{N}(x_i^{(8)})^{-3}\sum_{n=1}^{N}(s_i(n) - \hat{b}_i)^3$
Kurtosis	$x_i^{(10)} = \frac{1}{N}(x_i^{(8)})^{-4}\sum_{n=1}^{N}(s_i(n) - \hat{b}_i)^4$
Fluctuation	$x_i^{(11)} = \frac{1}{N-1}\left\lvert\sum_{n=2}^{N}\operatorname{sgn}(s_i(n) - s_i(n-1))\right\rvert$

Where the Selected Features are Located?

- As a side product we gain insight on which areas of the brain were used for prediction.

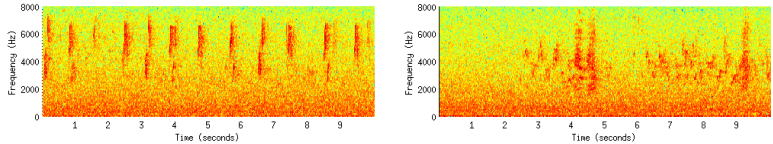


IEEE MLSP2013 Birds Competition

- The task of the participants was to train an algorithm to recognize *bird sounds* recorded at the wildlife.
- The recordings consisted of bird sounds from $C = 19$ species, and the sounds were overlapping (with up to 6 birds in a single recording).
- There were altogether 645 ten-second audio clips with sample rate $F_s = 16$ kHz.
- Half of the samples ($N_{\text{train}} = 323$) were used for model training, and half ($N_{\text{test}} = 322$) of the data was provided without annotation.

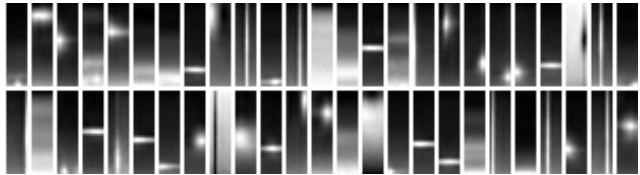
IEEE MLSP2013 Birds Competition

- The task of the participants was to train an algorithm to recognize *bird sounds* recorded at the wildlife.
- The recordings consisted of bird sounds from $C = 19$ species, and the sounds were overlapping (with up to 6 birds in a single recording).
- There were altogether 645 ten-second audio clips.
- Half of the samples were used for model training, and the participants should predict the labels of the other half.
- We participated in the challenge, and our final submission placed 7th among the 77 teams.



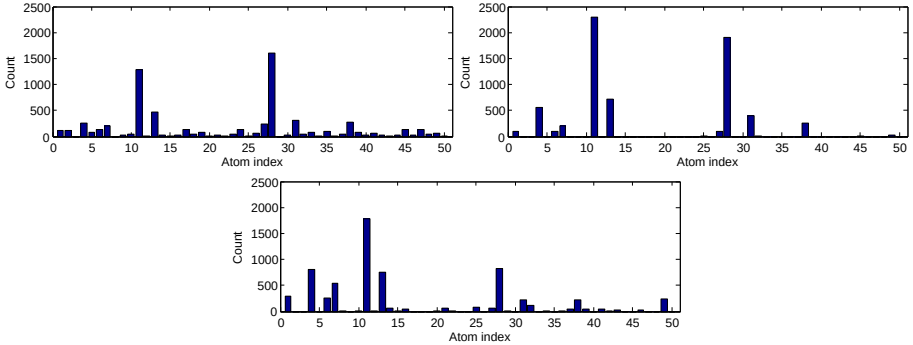
IEEE MLSP2013 Birds — Our Method

- Our approach mixed six prediction models by averaging the predicted class probabilities together.
- Most fruitful model used dictionary based Bag-of-Words features.
  - Learn a dictionary of typical patches in the spectrograms (including the non-annotated ones).
  - Count the occurrences of each dictionary atom in each spectrogram.
  - For dictionaly learning, use SPAMS: **http://spams-devel.gforge.inria.fr/**
  - Python and Matlab interfaces available.



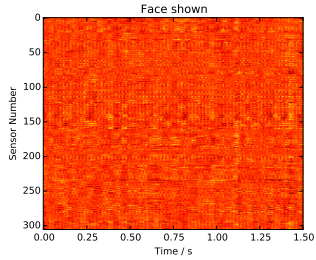
IEEE MLSP2013 Birds — Our Method

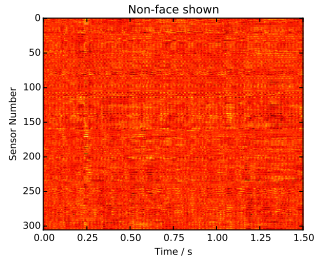
- Examples of patch histograms are shown below.
- Left: two bird species present, Right: no birds present, Bottom: a single bird present.
- One can spot the noise encoding patches (1, 4, 6, 7, 11 . . .), which are active in all histograms



DecMeg2014 Competition

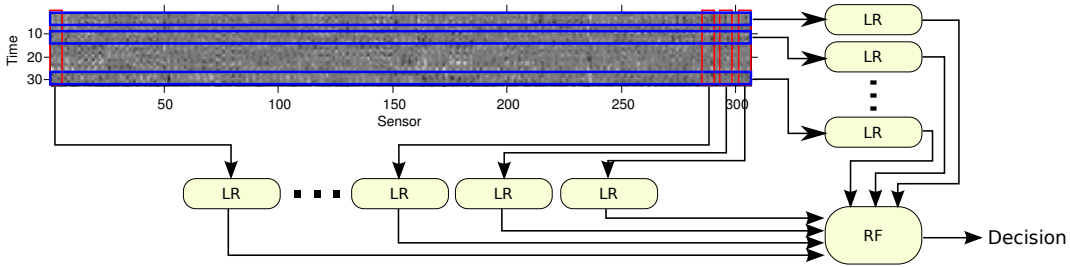
- In the 2014 DecMeg competition, the task was to predict whether test persons were shown a **face** or a **non-face** using MEG recordings of the brain.
- Sequences were 1.5 second long measurements with 306 channels; face shown at 0.5 s - 1.5 s.
- In total, 16 training and 7 test subjects; approximately 600 datapoints from each subject.
- Intersubject variation is significant.





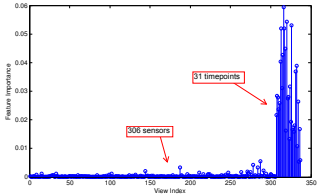
Our Model

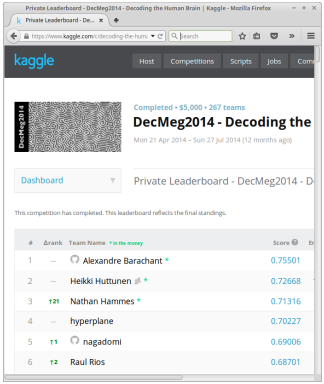
- Our is a hierarchical combination of two layers of classifiers:
  - First layer consists of Logistic Regression classifiers mapping individual sensors or individual timepoints to class probabilities.
  - The second layer is a 1000-tree Random Forest with LR predictions as inputs.
- The idea of stacking classifiers originates to Wolpert's *stacked generalization* (1992), and was inspired by our first attempts to approach the problem using a deep convolutional net.



Feature Importances

- The reliability of each 1st layer classifier can be seen from the feature importances of the Random Forest.
- The full time series from individual sensors seem a lot more informative than snapshots of the whole brain.
- Eventually our method was successful placing us on the 2nd place of 267 teams.
- The implementation is 100 % `sklearn`: **https://github.com/mahehu/decmeg**





Summary

- The significance of Python as a language of scientific computing has exploded.
- Machine learning has been one of the key areas in this development.
- This has been coined by the open-access attitude of the community: licensing tends to be very free.

- One of the key benefits of `scikit-learn` is the uniform API: It's easy to test a wide range of algorithms with a short piece of code.
- Deep learning is a growing trend—powered by Python.