W4995 Applied Machine Learning

Imputation and Feature Selection

02/13/17

Andreas Müller

Dealing with missing values

- Missing values can be encoded in many ways
- Numpy has no standard format for it (often np.NaN)
- Sometimes: 999, ???, ?, np.inf, “N/A”, “Unknown“ …
- Not discussing “missing output” - that’s semi-supervised learning.

```
array([[ nan,  3.2,  5.7,  2.3],
       [ nan,  2.8,  4.9,  2. ],
       [ nan,  2.8,  6.7,  2. ],
       [ nan,  2.7,  4.9,  1.8],
       [ 6.7,  3.3,  5.7,  2.1],
       [ nan,  3.2,  6. ,  1.8],
       [ nan,  2.8,  4.8,  1.8],
       [ nan,  3. ,  4.9,  1.8],
       [ nan,  2.8,  5.6,  2.1],
       [ nan,  3. ,  5.8,  1.6],
       [ 7.4,  2.8,  6.1,  1.9],
       [ nan,  3.8,  6.4,  2. ],
       [ 6.4,  2.8,  5.6,  2.2],
       [ nan,  2.8,  5.1,  1.5],
       [ nan,  2.6,  5.6,  1.4],
       [ nan,  3. ,  6.1,  2.3],
       [ nan,  3.4,  5.6,  2.4],
       [ nan,  3.1,  5.5,  1.8],
       [ nan,  3. ,  4.8,  1.8],
       [ 6.9,  3.1,  5.4,  2.1],
       [ 6.7,  3.1,  5.6,  2.4],
       [ nan,  3.1,  5.1,  2.3],
       [ nan,  2.7,  5.1,  1.9],
       [ nan,  3.2,  5.9,  2.3],
       [ nan,  3.3,  5.7,  2.5],
       [ nan,  3. ,  5.2,  2.3],
       [ nan,  2.5,  5. ,  1.9],
       [ nan,  3. ,  5.2,  2. ],
       [ 6.2,  3.4,  5.4,  2.3],
       [ nan,  3. ,  5.1,  1.8]])
```

```
array([[ 5.1,  3.5,  1.4,  0.2],
       [ nan,  nan,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ 4.6,  3.1,  1.5,  0.2],
       [ 5. ,  3.6,  1.4,  0.2],
       [ nan,  nan,  nan,  nan],
       [ 4.6,  3.4,  1.4,  0.3],
       [ 5. ,  3.4,  1.5,  0.2],
       [ 4.4,  2.9,  1.4,  0.2],
       [ 4.9,  3.1,  1.5,  0.1],
       [ 5.4,  3.7,  1.5,  0.2],
       [ 4.8,  3.4,  1.6,  0.2],
       [ 4.8,  3. ,  1.4,  0.1],
       [ 4.3,  3. ,  1.1,  0.1],
       [ nan,  nan,  nan,  nan],
       [ 5.7,  4.4,  1.5,  0.4],
       [ 5.4,  3.9,  1.3,  0.4],
       [ 5.1,  3.5,  1.4,  0.3],
       [ 5.7,  3.8,  1.7,  0.3],
       [ 5.1,  3.8,  1.5,  0.3],
       [ 5.4,  3.4,  1.7,  0.2],
       [ 5.1,  3.7,  1.5,  0.4],
       [ 4.6,  3.6,  1. ,  0.2],
       [ 5.1,  nan,  nan,  nan],
       [ 4.8,  3.4,  1.9,  0.2],
       [ 5. ,  3. ,  1.6,  0.2],
       [ nan,  nan,  nan,  0.4],
       [ 5.2,  3.5,  1.5,  0.2],
       [ 5.2,  3.4,  1.4,  0.2],
       [ 4.7,  3.2,  1.6,  0.2]])
```

Problem:
prediction not for all samples!
Accuracy might be biased!

```
array([[ 6. ,  3.4,  4.5,  nan],
       [ 6.9,  3.1,  5.1,  2.3],
       [ 4.6,  3.2,  1.4,  nan],
       [ 5.1,  3.8,  1.5,  nan],
       [ 4.4,  2.9,  nan,  nan],
       [ 6.6,  2.9,  4.6,  1.3],
       [ 6.7,  3. ,  5.2,  2.3],
       [ 6.3,  3.3,  6. ,  2.5],
       [ 7.2,  3. ,  5.8,  1.6],
       [ 4.6,  3.4,  1.4,  nan],
       [ 5.2,  3.5,  1.5,  nan],
       [ 5.4,  3.4,  nan,  nan],
       [ 5.9,  3.2,  4.8,  1.8],
       [ 4.9,  3.1,  nan,  nan],
       [ 6.9,  3.2,  5.7,  2.3],
       [ 5.7,  3.8,  nan,  0.3],
       [ 5.3,  3.7,  1.5,  nan],
       [ 4.5,  2.3,  1.3,  nan],
       [ 6.5,  3. ,  5.5,  1.8],
       [ 6.2,  2.9,  4.3,  1.3],
       [ 6.4,  2.8,  5.6,  2.2],
       [ 6.1,  3. ,  4.6,  1.4],
       [ 6.2,  2.8,  4.8,  1.8],
       [ 4.9,  2.5,  4.5,  1.7],
       [ 6. ,  2.7,  5.1,  nan],
       [ 6.8,  3.2,  5.9,  2.3],
       [ 6. ,  2.9,  4.5,  1.5],
       [ 4.9,  2.4,  3.3,  1. ],
       [ 5.8,  2.7,  5.1,  nan],
       [ 5.5,  2.4,  3.8,  nan]])
```

Imputation

```
array([[ 6. ,  3.4,  4.5,  1.6],
       [ 6.9,  3.1,  5.1,  2.3],
       [ 4.6,  3.2,  1.4,  0.2],
       [ 5.1,  3.8,  1.5,  0.3],
       [ 4.4,  2.9,  1.4,  0.2],
       [ 6.6,  2.9,  4.6,  1.3],
       [ 6.7,  3. ,  5.2,  2.3],
       [ 6.3,  3.3,  6. ,  2.5],
       [ 7.2,  3. ,  5.8,  1.6],
       [ 4.6,  3.4,  1.4,  0.3],
       [ 5.2,  3.5,  1.5,  0.2],
       [ 5.4,  3.4,  1.5,  0.4],
       [ 5.9,  3.2,  4.8,  1.8],
       [ 4.9,  3.1,  1.5,  0.1],
       [ 6.9,  3.2,  5.7,  2.3],
       [ 5.7,  3.8,  1.7,  0.3],
       [ 5.3,  3.7,  1.5,  0.2],
       [ 4.5,  2.3,  1.3,  0.3],
       [ 6.5,  3. ,  5.5,  1.8],
       [ 6.2,  2.9,  4.3,  1.3],
       [ 6.4,  2.8,  5.6,  2.2],
       [ 6.1,  3. ,  4.6,  1.4],
       [ 6.2,  2.8,  4.8,  1.8],
       [ 4.9,  2.5,  4.5,  1.7],
       [ 6. ,  2.7,  5.1,  1.6],
       [ 6.8,  3.2,  5.9,  2.3],
       [ 6. ,  2.9,  4.5,  1.5],
       [ 4.9,  2.4,  3.3,  1. ],
       [ 5.8,  2.7,  5.1,  1.9],
       [ 5.5,  2.4,  3.8,  1.1]])
```

Imputation methods

- Mean / median
- KNN
- Model-driven
- Iterative

```
from sklearn.linear_model import LogisticRegressionCV
from sklearn.model_selection import train_test_split, cross_val_score
X_train, X_test, y_train, y_test = train_test_split(X_, y, stratify=y, random_state=0)
```

```
nan_columns = np.any(np.isnan(X_train), axis=0)
X_drop_columns = X_train[:, ~nan_columns]
scores = cross_val_score(LogisticRegressionCV(cv=5), X_drop_columns, y_train, cv=10)
np.mean(scores)
```

```
0.77166666666666672
```

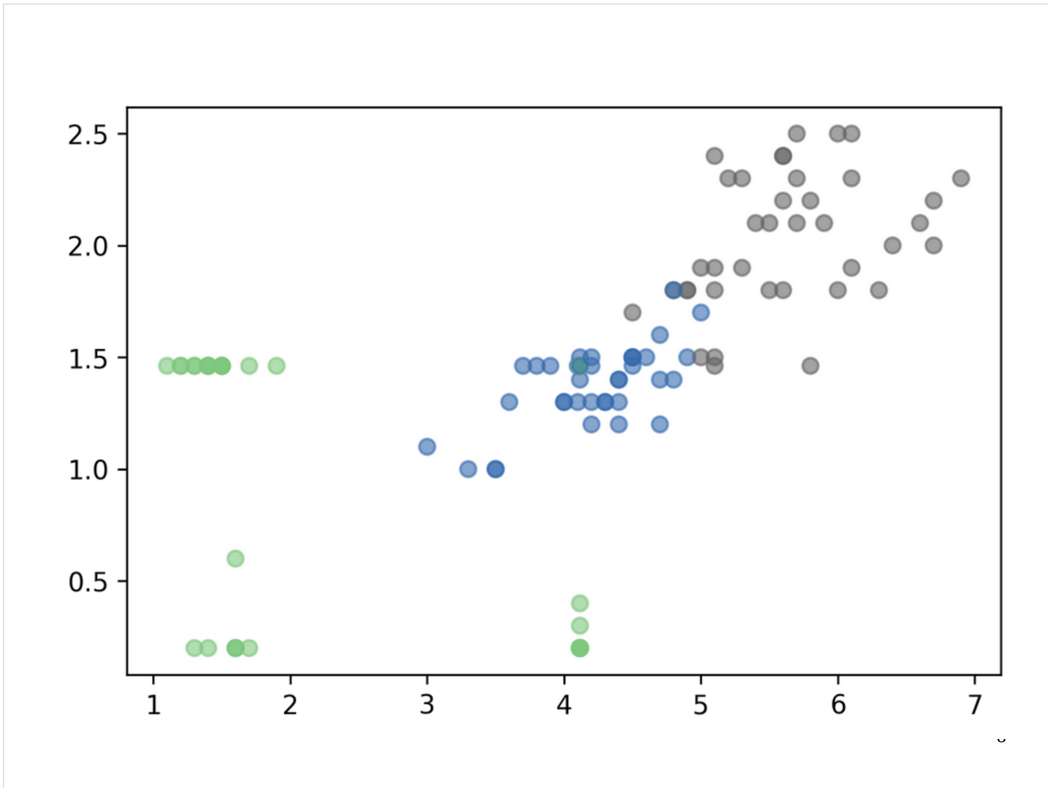
Mean and Median

```
[[ 6.   2.9  4.5  1.5]
 [ 5.9  3.   5.1  1.8]
 [ 4.4  3.   1.3  0.2]
 [ 5.1  3.3  nan  nan]
 [ 5.   3.5  1.6  0.6]
 [ 5.4  3.4  nan  nan]
 [ 5.7  3.8  nan  0.3]
 [ 5.6  2.5  3.9  nan]
 [ 7.7  2.6  6.9  2.3]
 [ 5.8  2.7  5.1  1.9]
 [ 6.7  3.1  5.6  2.4]
 [ 4.8  3.4  1.9  nan]
 [ 7.2  3.2  6.   1.8]
 [ 4.4  2.9  nan  nan]
 [ 6.9  3.2  5.7  2.3]
 [ 5.5  4.2  1.4  nan]
 [ 6.3  2.3  4.4  1.3]
 [ 7.   3.2  4.7  1.4]
 [ 5.8  2.7  nan  nan]
 [ 6.8  2.8  4.8  1.4]
 [ 5.4  3.9  1.7  nan]
 [ 7.6  3.   6.6  2.1]
 [ 7.7  2.8  6.7  2. ]
 [ 5.   3.3  nan  0.2]
 [ 5.9  3.   4.2  1.5]
 [ 6.1  2.8  4.   1.3]
 [ 5.   3.6  1.4  0.2]
 [ 7.4  2.8  6.1  1.9]
 [ 6.3  2.5  5.   1.9]
 [ 6.7  3.3  5.7  2.5]]
```

```python
from sklearn.preprocessing import Imputer
imp = Imputer(strategy="mean").fit(X_train)
imp.transform(X_train)[-30:]
```

Imputation

```
array([[ 6.   ,  2.9  ,  4.5  ,  1.5  ],
       [ 5.9  ,  3.   ,  5.1  ,  1.8  ],
       [ 4.4  ,  3.   ,  1.3  ,  0.2  ],
       [ 5.1  ,  3.3  ,  4.116,  1.462],
       [ 5.   ,  3.5  ,  1.6  ,  0.6  ],
       [ 5.4  ,  3.4  ,  4.116,  1.462],
       [ 5.7  ,  3.8  ,  4.116,  0.3  ],
       [ 5.6  ,  2.5  ,  3.9  ,  1.462],
       [ 7.7  ,  2.6  ,  6.9  ,  2.3  ],
       [ 5.8  ,  2.7  ,  5.1  ,  1.9  ],
       [ 6.7  ,  3.1  ,  5.6  ,  2.4  ],
       [ 4.8  ,  3.4  ,  1.9  ,  1.462],
       [ 7.2  ,  3.2  ,  6.   ,  1.8  ],
       [ 4.4  ,  2.9  ,  4.116,  1.462],
       [ 6.9  ,  3.2  ,  5.7  ,  2.3  ],
       [ 5.5  ,  4.2  ,  1.4  ,  1.462],
       [ 6.3  ,  2.3  ,  4.4  ,  1.3  ],
       [ 7.   ,  3.2  ,  4.7  ,  1.4  ],
       [ 5.8  ,  2.7  ,  4.116,  1.462],
       [ 6.8  ,  2.8  ,  4.8  ,  1.4  ],
       [ 5.4  ,  3.9  ,  1.7  ,  1.462],
       [ 7.6  ,  3.   ,  6.6  ,  2.1  ],
       [ 7.7  ,  2.8  ,  6.7  ,  2.   ],
       [ 5.   ,  3.3  ,  4.116,  0.2  ],
       [ 5.9  ,  3.   ,  4.2  ,  1.5  ],
       [ 6.1  ,  2.8  ,  4.   ,  1.3  ],
       [ 5.   ,  3.6  ,  1.4  ,  0.2  ],
       [ 7.4  ,  2.8  ,  6.1  ,  1.9  ],
       [ 6.3  ,  2.5  ,  5.   ,  1.9  ],
       [ 6.7  ,  3.3  ,  5.7  ,  2.5  ]])
```


2.5
2.0
1.5
1.0
0.5
1
2
3
4
5
6
7

```
from sklearn.linear_model import LogisticRegressionCV
from sklearn.model_selection import train_test_split, cross_val_score
X_train, X_test, y_train, y_test = train_test_split(X_, y, stratify=y, random_state=0)
```

```
nan_columns = np.any(np.isnan(X_train), axis=0)
X_drop_columns = X_train[:, ~nan_columns]
scores = cross_val_score(LogisticRegressionCV(cv=5), X_drop_columns, y_train, cv=10)
np.mean(scores)
```

```
0.77166666666666672
```

```
from sklearn.pipeline import make_pipeline
mean_pipe = make_pipeline(Imputer(), LogisticRegression())
scores = cross_val_score(mean_pipe, X_train, y_train, cv=10)
np.mean(scores)
```

```
0.95555555555555549
```

KNN imputation

- Find k nearest neighbors that have non-missing values.
- Fill in all missing values using the average of the neighbors.
- Different strategies possible for finding neighbors that have missing values.
- Tricky if there is no feature that is always non-missing.
- PR in scikit-learn: https://github.com/scikit-learn/scikit-learn/pull/4844

KNN imputation

```
from sklearn.neighbors import KNeighborsRegressor

# imput feature 2 with KNN
feature2_missing = np.isnan(X_train[:, 2])
knn_feature2 = KNeighborsRegressor().fit(X_train[~feature2_missing, :2],
                                         X_train[~feature2_missing, 2])

X_train_knn2 = X_train.copy()
X_train_knn2[feature2_missing, 2] = knn_feature2.predict(X_train[feature2_missing, :2])
```

```
# impute feature 3 with KNN
feature3_missing = np.isnan(X_train[:, 3])
knn_feature3 = KNeighborsRegressor().fit(X_train[~feature3_missing, :2],
                                         X_train[~feature3_missing, 3])

X_train_knn3 = X_train_knn2.copy()
X_train_knn3[feature3_missing, 3] = knn_feature3.predict(X_train[feature3_missing, :2])
```

An efficient implementation would find nearest neighbors only once!
This approach is quite naive as is requires the first two features to always be present!

Model-Driven Imputation

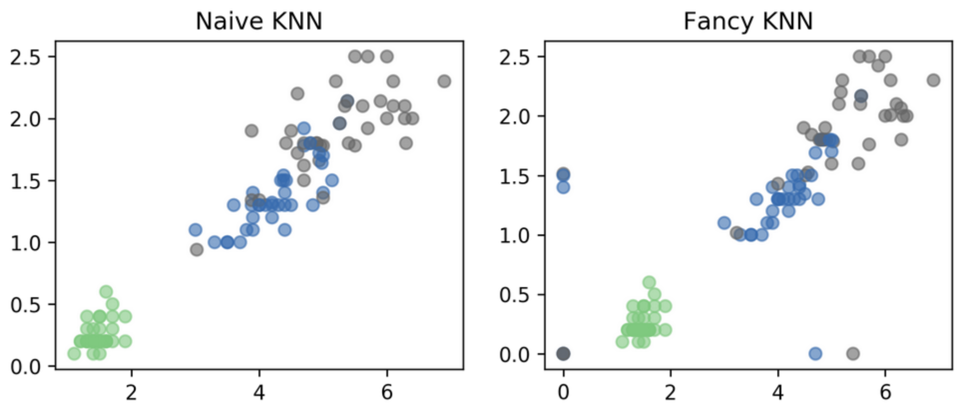
- Train regression model for missing values
- Possibly iterate: retrain after filling in
- Very flexible!

Model-driven Imputation w RF

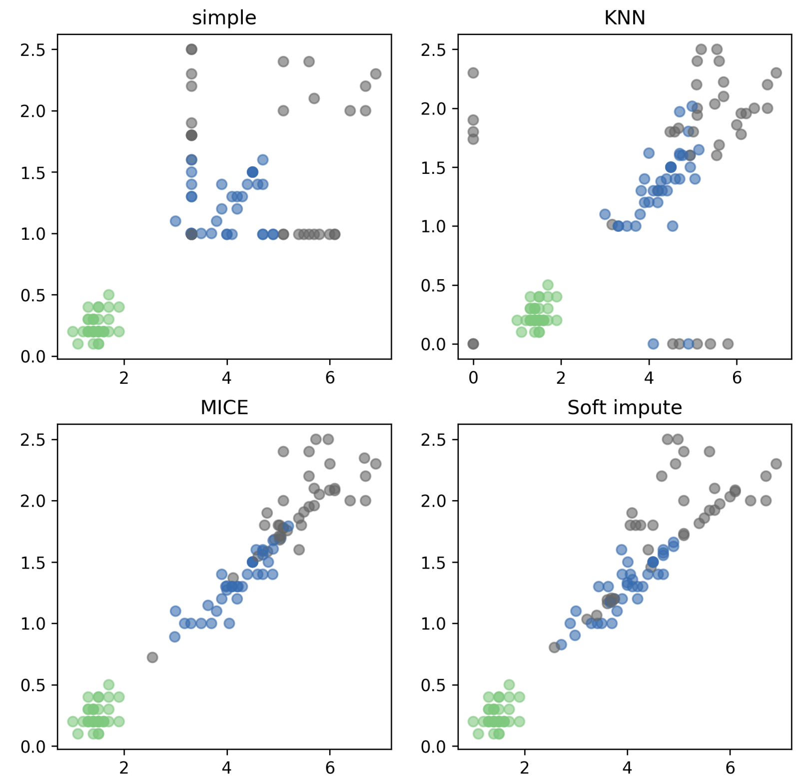
```
rf = RandomForestRegressor(n_estimators=100)

for i in range(10):
    last = X_imputed.copy()
    # imput feature 2 with rf

    rf.fit(X_imputed[~feature2_missing][:, inds_not_2], X_train[~feature2_missing, 2])

    X_imputed[feature2_missing, 2] = rf.predict(X_imputed[feature2_missing][:, inds_not_2])

    # impute feature 3 with rf

    rf.fit(X_imputed[~feature3_missing][:, inds_not_3], X_train[~feature3_missing, 3])
    X_imputed[feature3_missing, 3] = rf.predict(X_imputed[feature3_missing][:, inds_not_3])

    # this would make more sense if we scaled the data beforehand
    if (np.linalg.norm(last - X_imputed)) < .5:
        break
```

Fanyimpute

- pip install fancyimpute
- Has many methods – but no fit-transform paradigm
- MICE is iterative and works well often
- Try different things in practice, MICE might be best



Naive KNN
Fancy KNN


simple
KNN
MICE
Soft impute

Applying fancyimpute

```
X_train_fancy_mice = fancyimpute.MICE(verbose=0).complete(X_train)
scores = cross_val_score(logreg, X_imputed, y_train, cv=10)
scores.mean()
```

```
0.84833333333333327
```

Again, cheating with the imputation :(
This is allowed for the homework because the current tools make it hard to do the right thing.

Feature Selection

Why Select Features?

- Avoid overfitting
- Faster prediction and training
- Less storage for model and dataset

Types of Feature Selection

- Unsupervised vs Supervised
- Univariate vs Multivariate
- Model-based or not

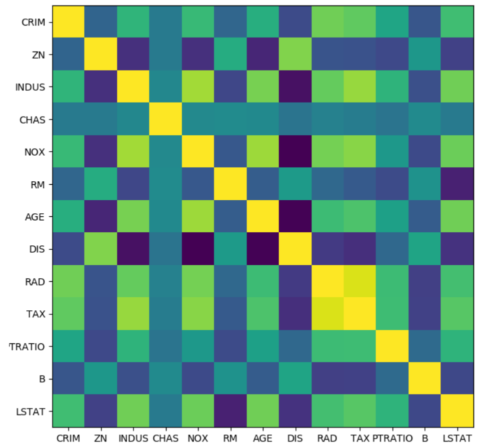
Unsupervised feature selection

- May discard important information
- Variance-based: 0 variance or few unique values
- Covariance-based: remove correlated features
- PCA: remove linear subspaces

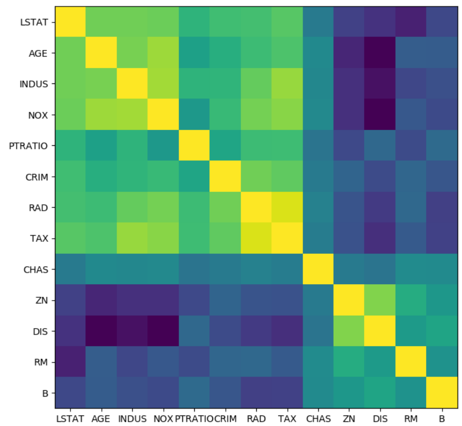
Covariance

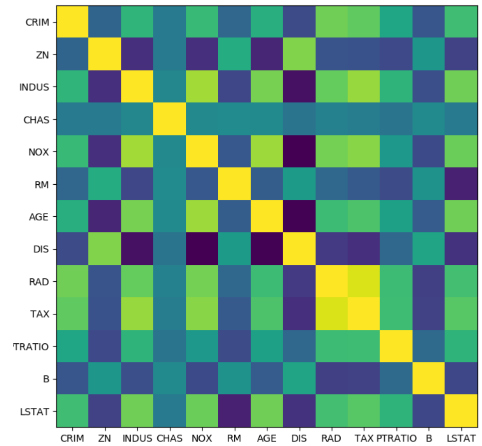
```
boston = load_boston()
X, y = boston.data, boston.target
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

```
from sklearn.preprocessing import scale
X_train_scaled = scale(X_train)
cov = np.cov(X_train_scaled, rowvar=False)
```



```python
from scipy.cluster import hierarchy
order = np.array(hierarchy.dendrogram(hierarchy.ward(cov), no_plot=True)['ivl'], dtype="int")
```





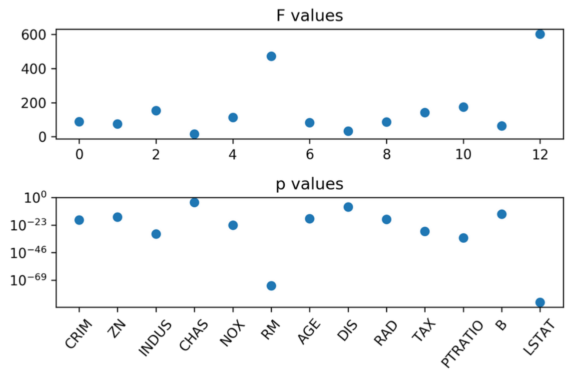
Can use this to iteratively remove most correlated pairs.
Beware: might discard important information!

Supervised Feature Selection

Simple univariate selection

- Pick statistic, check p-values!
- f_regression, f_classif, chi2 in scikit-learn

```
from sklearn.feature_selection import f_regression
f_values, p_values = f_regression(X, y)
```



```python
from sklearn.feature_selection import SelectKBest, SelectPercentile, SelectFpr
from sklearn.linear_model import RidgeCV

select = SelectKBest(k=2, score_func=f_regression)
select.fit(X_train, y_train)
print(X_train.shape)
print(select.transform(X_train).shape)
```

```
(379, 13)
(379, 2)
```

```python
all_features = make_pipeline(StandardScaler(), RidgeCV())
select_2 = make_pipeline(StandardScaler(), SelectKBest(k=2, score_func=f_regression), RidgeCV())
```

```python
np.mean(cross_val_score(all_features, X_train, y_train))
```

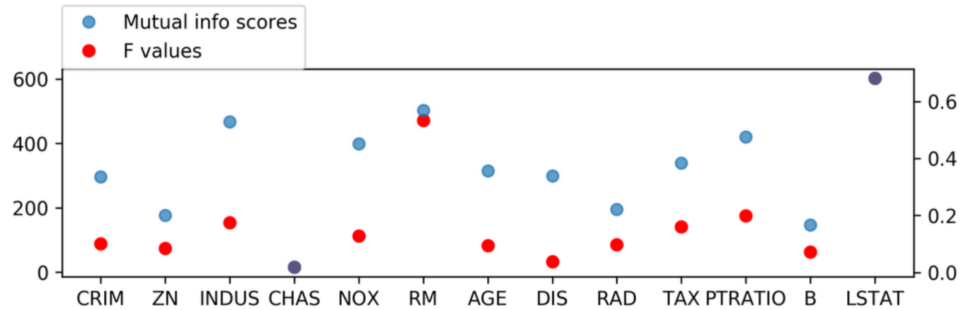
```
0.70430367076024736
```

```python
np.mean(cross_val_score(select_2, X_train, y_train))
```

```
0.63293351088220251
```

Mutual Information

```
from sklearn.feature_selection import mutual_info_regression
scores = mutual_info_regression(X_train, y_train, discrete_features=[3])
```



Mutual information (as implemented here) is also univariate,
but doesn't assume a linear model (like the F statistics do)
Can be used with SelectKBest etc

Model-Based Feature selection

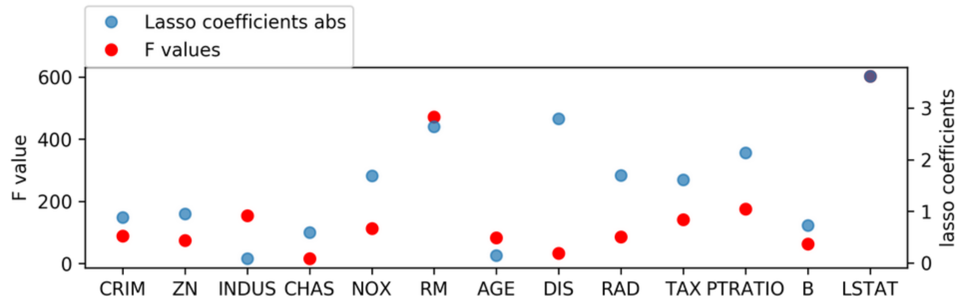
- Get best fit for a particular model
- Ideally: exhaustive search over all possible combinations
- Exhaustive is infeasible (and has multiple testing issues)
- Use heuristics in practice.

Model based (single fit)

- Build a model, select features most important to model.
- Lasso, other linear models, tree-based models
- Multivariate – linear models assume linear relation

```
from sklearn.linear_model import LassoCV
X_train_scaled = scale(X_train)
lasso = LassoCV().fit(X_train_scaled, y_train)
print(lasso.coef_)
```

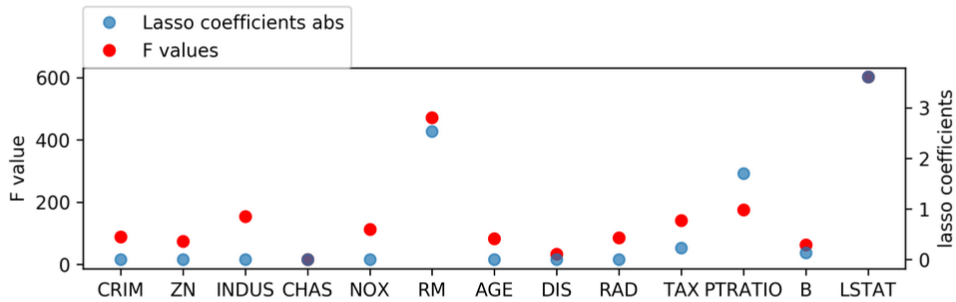
```
[-0.881  0.951 -0.082  0.59  -1.69   2.639 -0.146 -2.796  1.695 -1.614
 -2.133  0.729 -3.615]
```



Changing Lasso alpha

```
from sklearn.linear_model import Lasso
X_train_scaled = scale(X_train)
lasso = Lasso().fit(X_train_scaled, y_train)
print(lasso.coef_)
```

```
[-0.     0.    -0.     0.    -0.     2.529 -0.    -0.    -0.    -0.228
 -1.701  0.132 -3.606]
```



SelectFromModel

```
from sklearn.feature_selection import SelectFromModel
select_lassocv = SelectFromModel(LassoCV(), threshold="median")
select_lassocv.fit(X_train, y_train)
print(select_lassocv.transform(X_train).shape)
```

```
(379, 7)
```

```
pipe_lassocv = make_pipeline(StandardScaler(), select_lassocv, RidgeCV())
np.mean(cross_val_score(pipe_lassocv, X_train, y_train))
```

```
0.69547599764583534
```

```
np.mean(cross_val_score(all_features, X_train, y_train))
```

```
0.70430367076024736
```

```
# could grid-search alpha in lasso
select_lasso = SelectFromModel(Lasso())
pipe_lasso = make_pipeline(StandardScaler(), select_lasso, RidgeCV())
np.mean(cross_val_score(pipe_lasso, X_train, y_train))
```

```
0.66651484975652764
```

Iterative Model-Based Selection

- Fit model, find least important feature, remove, iterate.

- Or: Start with single feature, find most important feature, add, iterate.

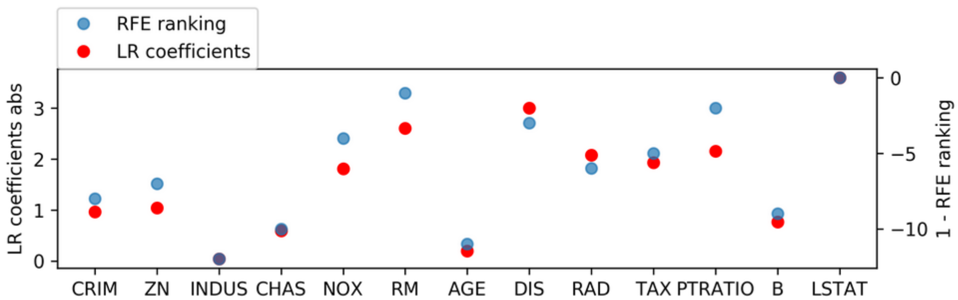
Recursive Feature Elimination

- Uses feature importances / coefficients, similar to "SelectFromModel"
- Iteratively removes features (one by one or in groups)
- Runtime:

  (n_features - n_feature_to_keep) / stepsize

```python
from sklearn.linear_model import LinearRegression
from sklearn.feature_selection import RFE

# create ranking among all features by selecting only one
rfe = RFE(LinearRegression(), n_features_to_select=1)
rfe.fit(X_train_scaled, y_train)
rfe.ranking_
```

```
array([ 9,  8, 13, 11,  5,  2, 12,  4,  7,  6,  3, 10,  1])
```



RFECV

- Efficient CV for n_features_to_keep

```
from sklearn.linear_model import LinearRegression
from sklearn.feature_selection import RFECV

rfe = RFECV(RidgeCV())
rfe.fit(X_train_scaled, y_train)
print(rfe.support_)
print(boston.feature_names[rfe.support_])
```

```
[False False False False  True  True False  True False False  True False
  True]
['NOX' 'RM' 'DIS' 'PTRATIO' 'LSTAT']
```

```
pipe_rfe_ridgecv = make_pipeline(StandardScaler(), RFECV(RidgeCV()))
np.mean(cross_val_score(pipe_rfe_ridgecv, X_train, y_train, cv=10))
```

```
0.70961371790492289
```

If we want to predict with the same model as used for selection, RFECV can be used as the prediction step.
Could also use RFECV as transformer and use any other model!

```
from sklearn.preprocessing import PolynomialFeatures
pipe_rfe_ridgecv = make_pipeline(StandardScaler(), PolynomialFeatures(), RFECV(LinearRegression(), cv=10), RidgeCV())
np.mean(cross_val_score(pipe_rfe_ridgecv, X_train, y_train, cv=10))
```

```
0.82031507795394398
```

Wrapper Methods

- Can be applied for ANY model!
- Shrink / grow feature set by greedy search
- Called Forward or Backward selection
- Complexity: n_features * (n_features + 1) / 2
- Implemented in mlxtend

SequentialFeatureSelector

```
from mlxtend.feature_selection import SequentialFeatureSelector
sfs = SequentialFeatureSelector(LinearRegression(), forward=False, k_features=7)
sfs.fit(X_train_scaled, y_train)
```

```
Features: 7/7
```

```
SequentialFeatureSelector(clone_estimator=True, cv=5,
             estimator=LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False),
             floating=False, forward=False, k_features=7, n_jobs=1,
             pre_dispatch='2*n_jobs', print_progress=None, scoring=None,
             skip_if_stuck=True, verbose=1)
```

```
print(sfs.k_feature_idx_)
print(boston.feature_names[np.array(sfs.k_feature_idx_)])
```

```
(1, 4, 5, 7, 9, 10, 12)
['ZN' 'NOX' 'RM' 'DIS' 'TAX' 'PTRATIO' 'LSTAT']
```

```
sfs.k_score_
```

```
0.72508612228296487
```