assignment2-solution

March 14, 2017

1 Foundations of Data Mining: Assignment 2

Please complete all assignments in this notebook. You should submit this notebook, as well as a PDF version (See File > Download as).

In [1]:
```
%matplotlib inline
from preamble import *
plt.rcParams['savefig.dpi'] = 100
InteractiveShell.ast_node_interactivity = "all"


import matplotlib.pyplot as plt
import pandas as pd
import random
import xmltodict
from IPython.display import clear_output
from openml import runs, tasks
from sklearn import ensemble
from sklearn.datasets import make_blobs
from sklearn.decomposition import TruncatedSVD
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import (
    AdaBoostClassifier, GradientBoostingClassifier,
    RandomForestClassifier, RandomForestRegressor
)
from sklearn.feature_selection import (
    SelectFromModel, SelectKBest, chi2, f_regression
)
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import (RBF, ConstantKernel,
                                              WhiteKernel)
from sklearn.kernel_ridge import KernelRidge
from sklearn.linear_model import (LinearRegression, LogisticRegression,
                                  RidgeClassifier)
from sklearn.metrics import accuracy_score, r2_score
from sklearn.model_selection import (GridSearchCV, RandomizedSearchCV,
                                     ShuffleSplit, StratifiedKFold,
```

```
                                        cross_val_score, train_test_split)
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import (
    Imputer, OneHotEncoder, PolynomialFeatures,
    RobustScaler, MinMaxScaler
)
from sklearn.svm import SVC, SVR, LinearSVC
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.utils import shuffle
```

1.1 Kernel selection (4 points (1+2+1))

SVMs can be trained with different kernels. Generate a 2-dimensional dataset as shown below and study the effect of the choice of kernel by visualizing the results.

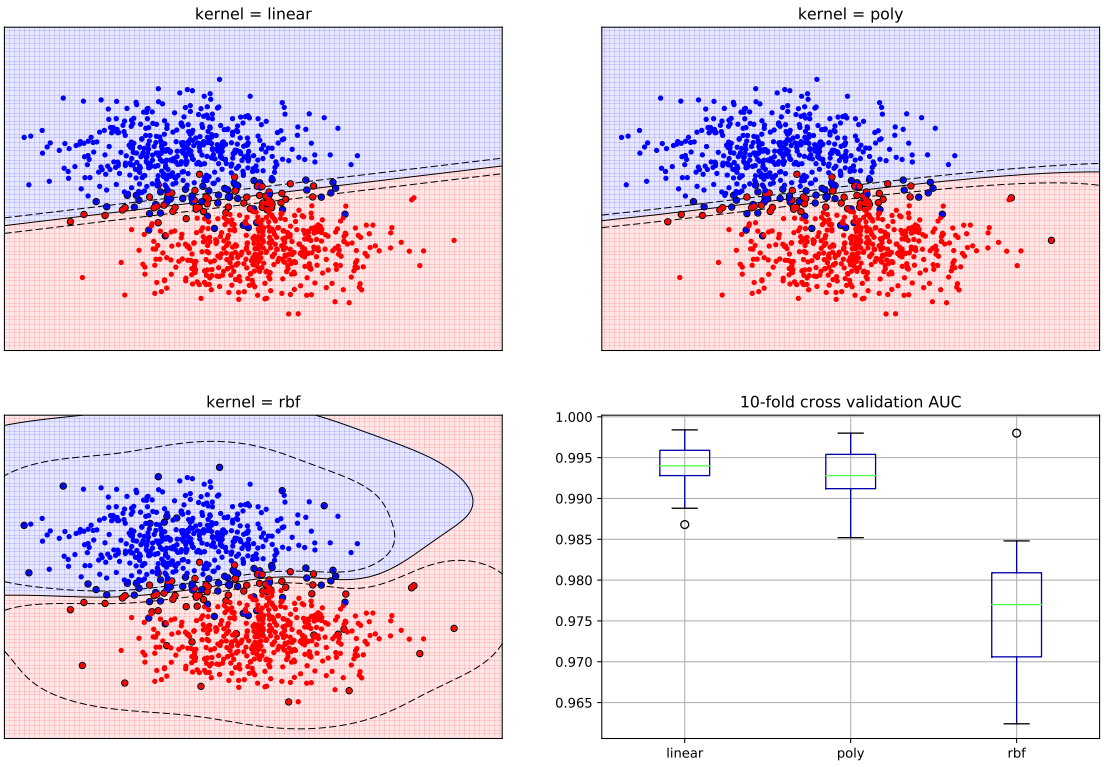
```
In [2]: X, y = make_blobs(centers=2, n_samples=1000, random_state=0)
```

```
def plot_svm(clf, support_vectors=True, colors=True,
             contours=True, plot_perc=1.0):
    if support_vectors:
        plt.scatter(
            clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
            s=85, edgecolors='k', c='w', zorder=10, marker='.'
        );

    if plot_perc > 0.0:
        if plot_perc < 1.0:
            num = int(plot_perc * X.shape[0])
            indexes = np.array(
                np.floor(np.random.sample(X.shape[0]) * num),
                dtype=np.int
            )
            xs, ys, cs = X[indexes, 0], X[indexes, 1], y[indexes]
        else:
            xs, ys, cs = X[:,0], X[:,1], y

        plt.scatter(xs, ys, c=cs, zorder=10, cmap=plt.cm.bwr, marker='.');

    plt.axis('tight');
    x_min = -2
    y_min = -3
    x_max = 6
    y_max = 9

    XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
```

```
    Z = clf.decision_function(np.c_[XX.ravel(), YY.ravel()]);

    Z = Z.reshape(XX.shape);
    if colors:
        plt.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.bwr, alpha=0.1);

    if contours:
        plt.contour(XX, YY, Z, colors=['k', 'k', 'k'],
                    linestyles=['--', '-', '--'],
                    levels=[-.5, 0, .5]);

    plt.xlim(x_min, x_max); plt.ylim(y_min, y_max);
    plt.xticks([]); plt.yticks([]);
```

- Train a SVM classifier on the dataset using respectively a linear, polynomial and radial basis function (RBF) kernel, evaluate the performance of each kernel using 10-fold cross-validation and AUC. Which one works best? Visualize the results. Can you intuitively explain why one kernel is more suited than another?
  - Hint: you can use the visualization code used in class. It is under mglearn/plot_svm.py > plot_svm_kernels().

```
In [141]: scores = {}
          plt.figure(figsize=(15, 10));
          for i, kernel in enumerate(['linear', 'poly', 'rbf']):
              clf = SVC(kernel=kernel);
              scores[kernel] = cross_val_score(clf, X, y, cv=10,
                                               scoring='roc_auc');
              clf.fit(X, y);

              #plt.figure();
              plt.subplot(2, 2, i + 1);
              plt.title('kernel = %s' % kernel);
              plot_svm(clf)


          #plt.figure();
          plt.subplot(2, 2, 4);
          df = pd.DataFrame(list(scores.values()), index=list(scores.keys()));
          df.T.boxplot(list(scores.keys()));
          plt.title('10-fold cross validation AUC')
          plt.show();
```

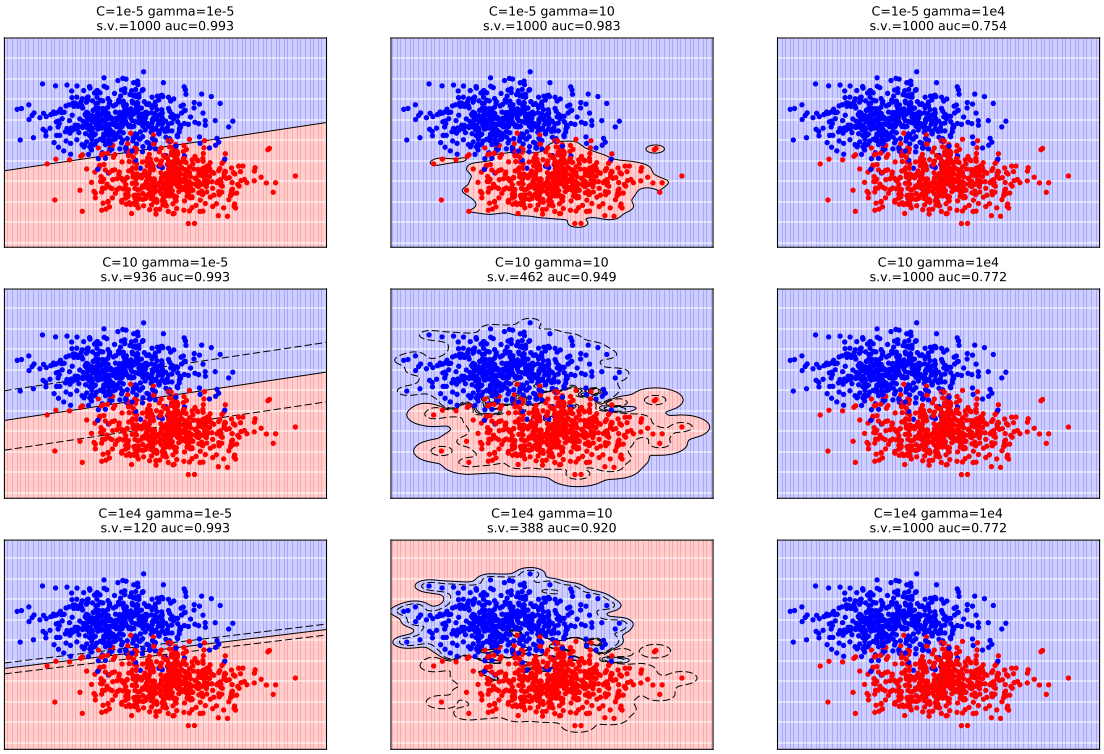


The linear and polynomial kernels work best in this scenario, and they produce very similar classification boundaries. It seems like the optimal separation boundary between the two blobs is actually linear, which can be modeled well by those two kernels, while the RBF kernel seems to produce a wawe-like separation boundary. This boundary has some variability which depends on the actual points used to build it, explaining why the RBF kernel has slightly lower scores; in other words, it tends to overfit the data.

- Take the RBF kernel and vary both the C parameter and the kernel width (γ). Use 3 values for each (a very small, default, and very large value). For each of the 9 combinations, create the same RBF plot as before, report the number of support vectors, and the AUC performance. Explain the performance results. When are you over/underfitting?
  - Hint: values for C and γ are typically in $[2^{-15}..2^{15}]$ on a log scale.
  - Hint: don't count the support vectors manually, retrieve them from the trained SVM.

```
In [3]: import mglearn

        plt.figure(figsize=(15, 10));
        fig_num = 0
        for c in ['1e-5', '10', '1e4']:
            for gamma in ['1e-5', '10', '1e4']:
                svm = SVC(kernel='rbf', C=float(c), gamma=float(gamma))
                auc = cross_val_score(svm, X, y, cv=10, scoring='roc_auc')
                svm.fit(X, y)
```

```
            fig_num += 1
            plt.subplot(3, 3, fig_num);
            plt.title('C=%s gamma=%s\ns.v.=%d auc=%.3f' % (
                c, gamma, svm.support_vectors_.shape[0], auc.mean()
            ));
            plot_svm(svm, support_vectors=False);

    plt.show();
```

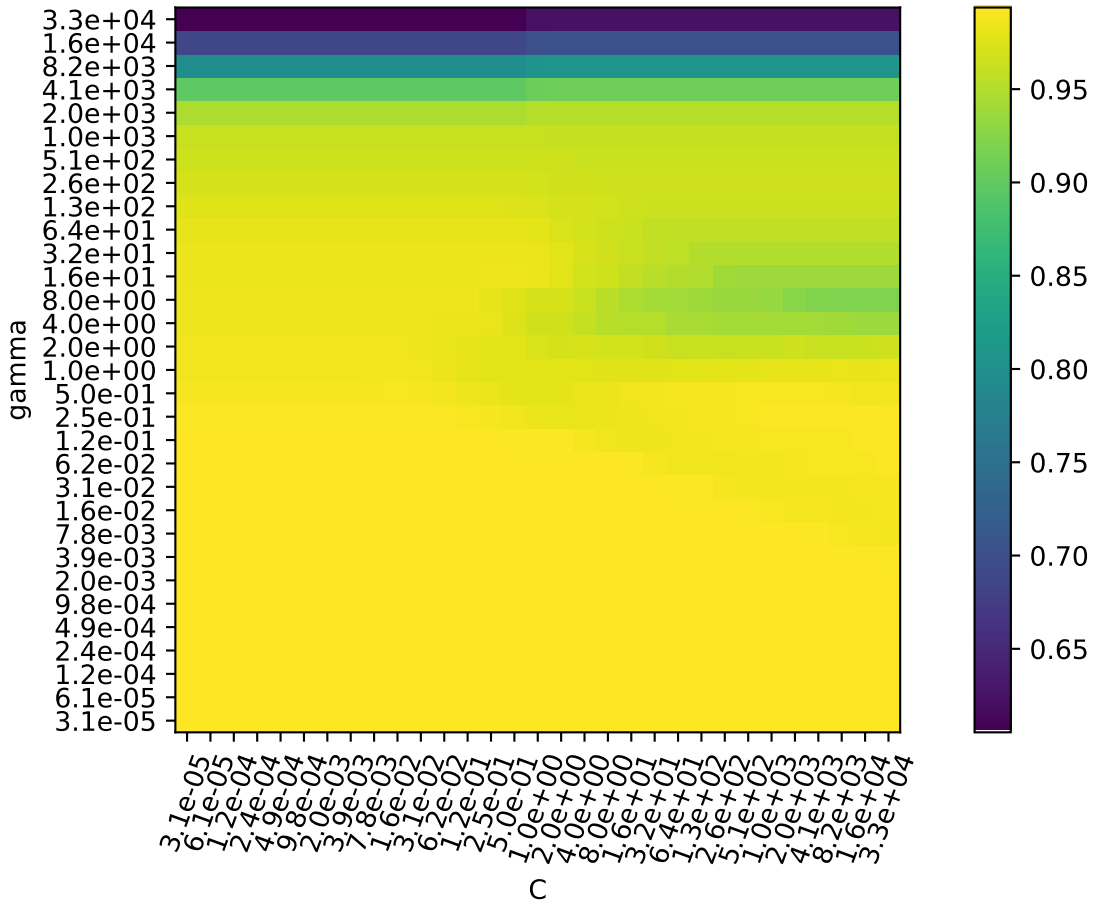


From these plots, it seems like high gamma values lead to overfitting, while low gamma values tend to models resemble a linear model; note that the plot with C=1e4 and gamma=1e-5 is almost identical to the one produced with the linear kernel in the previous section. As gamma increases, the separation boundary closes around the points of one class.

- Vary C and γ again, but this time use a grid of at least 20x20, vary both parameters uniformly on a log scale, and visualise the results using a $C \times \gamma \rightarrow AUC$ heatmap. Explain the performance results, and compare them to the 9 results obtained in the previous subquestion. Can you also tell in which regions of the heatmap you are over/underfitting?
  - Hint: We've constructed such a heatmap in class and in assignment 1.

```
In [101]: vals = [2**x for x in range(-15, 16)]

          param_grid = {
```

```
    'C': vals,
    'gamma': vals,
}

gs = GridSearchCV(SVC(kernel='rbf'), param_grid=param_grid,
                  cv=10, n_jobs=-1, scoring='roc_auc');
gs.fit(X, y);


scores = np.array(gs.cv_results_['mean_test_score']).reshape(
    len(param_grid['C']),
    len(param_grid['gamma'])
).T;

plt.figure(figsize=(10, 5))
img = mglearn.tools.heatmap(
    scores, cmap='viridis', fmt='',
    ylabel='gamma', yticklabels=param_grid['gamma'],
    xlabel='C', xticklabels=param_grid['C'],
);
plt.setp(img.get_axes().xaxis.get_majorticklabels(),
         rotation=70);
plt.xticks(plt.xticks()[0], ['%.1e' % x for x in vals]);
plt.yticks(plt.yticks()[0], ['%.1e' % x for x in vals]);
plt.colorbar(img);
```



Very high gamma values lead to a prediction boundary which basically wraps every point of one class in little "bubbles", hence the high overfitting shown in both plots above. The green-ish region located in the mid-right part of the matrix corresponds to the central plot drawn in the previous point; this seems to happen when the kernel starts to "wrap" one of the two blobs inside its own bubble. It only happens with medium to high values of C, meaning that there will be few support vectors in the region between the two blobs; this causes the boundary to be highly dependent on the training examples, and likely to miss the nearby test examples. Consitently with what revealed in the previous point, very high gamma values lead to models that wrap each training point with its own "bubble", with terrible test performances. On the other hand, the SVM never underfits, as a linear separation boudary is probably the best one, in this case (has an accuracy of 99.3%).

1.2 Robots and SVMs (4 points (2+1+1))

The Wall Robot Navigation dataset contains about 5500 readings of an ultrasound sensor array mounted on a robot, and your task is to finetune and train an SVM classifier to predict how the robot should move next.

```
In [32]: robot_data = oml.datasets.get_dataset(1497)
         X, y = robot_data.get_data(
```

```
    target=robot_data.default_target_attribute
);


def run_random_search(X, y, **kwargs):
    vals = [2**x for x in range(-8, 8)]
    param_grids = [
        {'kernel': ['linear'], 'C': vals},
        {'kernel': ['poly'], 'C': vals, 'degree': range(2, 11)},
        {'kernel': ['rbf'], 'C': vals, 'gamma': vals},
    ]

    best_estimator = best_params = best_score = None
    for grid in param_grids:
        rs = RandomizedSearchCV(
            SVC(), param_distributions=grid,
            scoring='accuracy', **kwargs
        );
        _ = rs.fit(X, y);
        if not best_score or rs.best_score_ > best_score:
            best_score = rs.best_score_
            best_params = rs.best_params_
            best_estimator = rs.best_estimator_

    return best_estimator, best_score, best_params
```

- Make a stratified 80-20 split of the data. On the training set alone, optimize the main hyperparameters of the SVM for Accuracy with a random search. Vary at least the main kernel types (linear, polynomial, and RBF), the C parameter, the γ parameter for the RBF kernel and the exponent/degree for the polynomial kernel. Report the optimal hyperparameter settings and Accuracy performance.
  - The degree of the polynonial is typically in the range 2..10.
  - Hint: note that the hyperparameter ranges depend on each other. For instance, γ only makes sense if you have selected the RBF kernel as well. We've seen in class how to define multiple hyperparameter spaces in a random/grid search.
- Train an SVM using the optimal hyperparameter configuration you found (in part 1 of this question) and test it on the held out (20%) test set. Compare this Accuracy result with the (mean) result of the nested CV. If you would build this robot in practice, how would you find the hyperparameters to use, and which performance would you expect? Is it truly necessary to tune the hyperparameters? Which hyperparameters were most important to tune?

```
In [34]: Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.2);
         best_estimator, best_score, best_params = run_random_search(
             Xtrain, ytrain, cv=5, n_jobs=-1, n_iter=15, verbose=3
         )

         clear_output()
```

```
        print('Best accuracy on training set: {:.3f} with parameters {}'.format(
            best_score, best_params
        ))

        test_accuracy = accuracy_score(ytest, best_estimator.predict(Xtest))
        print('Accuracy on the holdout test set: {:.3f}'.format(test_accuracy))
```

```
Best accuracy on training set: 0.921 with parameters {'kernel': 'rbf', 'gamma': 0.0
Accuracy on the holdout test set: 0.936
```

An SVM may too overpowered for handling the navigation of a robot, depending on the complexity of the robot itself and the environment it is expected to operate in. It would be ineffective to train the SVM using an existing dataset, since the position of the sensors might be different, there could be more or less sensors and so on; I would build a test environment and manually remote-control the robot through a number of different situations, to acquire enough sensor data and build a dataset.

Achieving the maximum possible classification performance wouldn't be so important, depending on the current performances. For example, an improvement from 80% (1 error every 5) to 85% (1 error every 7) won't have an effect as noticeable as an improvement from 94% (1 error every 17) to 99% (1 error every 100), and improvements below 1% would have no effects at all, unless performances are already very high and there are very strict requirements for the robot, in which case the SVM should be paired with another system. An estimate of this effect can be obtained with the combined mean accuracy score plus its variance (see the next point).

- Use a 5x3-fold (5 outer, 3 inner) nested cross-validation (CV) on the **whole** dataset to obtain a clean evaluation. What is the mean optimized performance? Is this in line with the optimized result of the random search of the previous question?

```
In [33]: scores = []
         for train_index, test_index in StratifiedKFold(5).split(X, y):
             estimator, _, _ = run_random_search(
                 X[train_index], y[train_index], cv=3,
                 n_jobs=-1, verbose=3, n_iter=15
             )
             scores.append(accuracy_score(
                 y[test_index], estimator.predict(X[test_index])
             ))

         clear_output()
         print('Mean accuracy: {:.3f}'.format(sum(scores) / len(scores)))
         print('Scores standard deviation: {:.3f}'.format(
             np.array(scores).std()
         ))
```

```
Mean accuracy: 0.866
Scores standard deviation: 0.048
```

As expected, the mean score of nested cross validation is slightly lower (after taking into account its standard deviation) than the score of the random search on the full training dataset. In fact, the latter is slightly biased, because it is computed on samples that were used for training in another iteration, while the score obtained with the former method is based on samples that were never seen during the optimization.

The variance is rather high, and the best model achieves less than half of the mistakes of the worst one (respectively one error every $[1-(0.866-0.048)]^{-1} = 5.5$ and one error every $[1-(0.866+0.048)]^{-1} = 11.6$), but one needs to consider that the models were trained with only $\frac{4}{5} \cdot \frac{2}{3} = \frac{8}{15}$ of the available data. Given the high difference between the scores of nested cross validation and the plain random search (trained with 90% of the data), I expect the final accuracy to be somewhere close to 0.9, when the SVM is trained with the full dataset,

1.3 A benchmark study (3 points (2+1))

A benchmark study is an experiment in which multiple algorithms are evaluated on multiple datasets. The end goal is to study whether one algorithm is generally better than the others. Meaningful benchmark studies can grow quite complex, here we do a simplified variant.

```
In [51]: def clean_dataset(dataset):
             pipeline_steps = []

             ds = oml.datasets.get_dataset(dataset);
             X, y = ds.get_data(target=ds.default_target_attribute);

             nominal = [
                 int(x['oml:index']) for x in ds.features['oml:feature']
                 if (x['oml:data_type'] == 'nominal'
                     and x['oml:is_target'] == 'false')
             ]
             numeric = [
                 int(x['oml:index']) for x in ds.features['oml:feature']
                 if (x['oml:data_type'] == 'numeric'
                     and x['oml:is_target'] == 'false')
             ]

             target = [
                 int(x['oml:index']) for x in ds.features['oml:feature']
                 if x['oml:is_target'] == 'true'
             ][0]
             nominal = [x if x < target else x - 1 for x in nominal]
             numeric = [x if x < target else x - 1 for x in numeric]

             if nominal:
                 fill = np.nanmax(X, axis=0) + 1
                 for col, val in enumerate(fill):
                     if col in nominal:
                         fix = np.logical_or(np.isnan(X[:, col]),
                                             np.isinf(X[:, col]))
```

```
                X[fix, col] = val
                assert (np.isfinite(X[:, col].sum())
                        and np.isfinite(X[:, col]).all(), X
                assert np.all(np.mod(X[:, col], 1) == 0)
            n_values = [np.unique(X[:, x]).shape[0] for x in nominal]
            enc = OneHotEncoder(n_values=n_values, sparse=False
                                categorical_features=nominal)
            pipeline_steps.append(('enc', enc))

        if numeric:
            pipeline_steps.append(('imputer', Imputer()))
            pipeline_steps.append(('scaler', RobustScaler()))

        identifier = [
            int(x['oml:index']) for x in ds.features['oml:feature']
            if x['oml:is_row_identifier'] == 'true'
        ]
        if identifier:
            idf = (identifier[0] if identifier[0] < target
                   else identifier[0] - 1)
            X = np.hstack([X[:, :idf], X[:, (idf+1):]])

        return X, y, pipeline_steps


    def test_estimator_on_dataset(estimator, dataset, params=None):
        X, y, pipeline_steps = clean_dataset(dataset)

        if params:
            pip = Pipeline(pipeline_steps + [
                ('reducer', SelectKBest()),
                ('est', estimator),
            ])
            params = {'est__' + k: v for k, v in params.items()}
            params['reducer__k'] = [
                int(X.shape[1] * x) for x in [0.25, 0.5, 0.75, 1.0]
            ]
            rs = GridSearchCV(pip, param_grid=params, n_jobs=-1, cv=10)
            rs.fit(X, y)
            return rs.best_score_
        else:
            pip = Pipeline(pipeline_steps + [('est', estimator)])
            scores = cross_val_score(pip, X, y, scoring='roc_auc',
                                     n_jobs=-1, cv=10)
            return scores.mean()
```

- Download OpenML datasets 37, 470, 1120, 1464 and 1471. They are sufficiently large (e.g., at least 500 data points) so that the performance estimation is trustworthy. Select at least

three classifiers that we discussed in class, e.g. kNN, Logistic Regression, Random Forests, Gradient Boosting, SVMs, Naive Bayes. Note that some of these algorithms take longer to train. Evaluate all classifiers (with default parameter settings) on all datasets, using a 10-fold CV and AUC. Show the results in a table and interpret them. Which is the best algorithm in this benchmark?

  - Note that these datasets have categorical features, different scales, missing values, and (likely) irrelevant features. You'll need to build pipelines to correctly build all models. Also remove any row identifiers (see, e.g., https://www.openml.org/d/1120)
  - Hint: You can either compare the performances directly, or (better) use a statistical significance test, e.g. a pairwise t-test or (better) Wilcoxon signed ranks test, to see whether the performance differences are significant. This is covered in statistics courses. You can then count wins, ties and losses.

- Repeat the benchmark, but now additionally optimize the main hyperparameters of each algorithm in a grid or random search (explore at least 5 values per hyperparameter, where possible). Does this affect the ranking of the algorithms?

```
In [73]: estimators = [
             ('knn', KNeighborsClassifier(), {
                 'n_neighbors': [1, 2, 3, 4, 5, 10, 15, 20, 50, 100]
             }),
             ('lr', LogisticRegression(), {
                 'C': [2**x for x in range(-16, 17)]
             }),
             ('rf', RandomForestClassifier(), {
                 'n_estimators': [2**x for x in range(0, 10)]
             }),
             ('gb', GradientBoostingClassifier(), {
                 'learning_rate': [2**x for x in range(-5, 1)],
                 'n_estimators': [500],
             }),
             ('gnb', GaussianNB(), None)
         ]

         datasets = [37, 470, 1120, 1464, 1471]

         fig = plt.figure(figsize=(15, 10));
         for i, ds in enumerate(datasets):
             scores = []
             for name, est, grid in estimators:
                 default_score = test_estimator_on_dataset(est, ds, None)
                 optimized_score = test_estimator_on_dataset(est, ds, grid)
                 scores.append((default_score, optimized_score))

             scores = pd.DataFrame(
                 scores, index=[name for name, _, _ in estimators],
                 columns=['Default', 'Optimized']
             );
```

```
            print('Dataset', ds)
            display(scores)
            sub = fig.add_subplot(2, 3, (i + 1) if i != 4 else 6);
            img = scores.T.plot(kind='bar', ax=sub, rot=0
                                title='Dataset %d' % ds)
        fig.show()
```

```
Dataset 37
```

	Default	Optimized
knn	0.78	0.77
lr	0.83	0.77
rf	0.80	0.77
gb	0.83	0.77
gnb	0.82	0.82

```
Dataset 470
```

	Default	Optimized
knn	0.61	0.68
lr	0.77	0.68
rf	0.61	0.65
gb	0.67	0.65
gnb	0.60	0.60

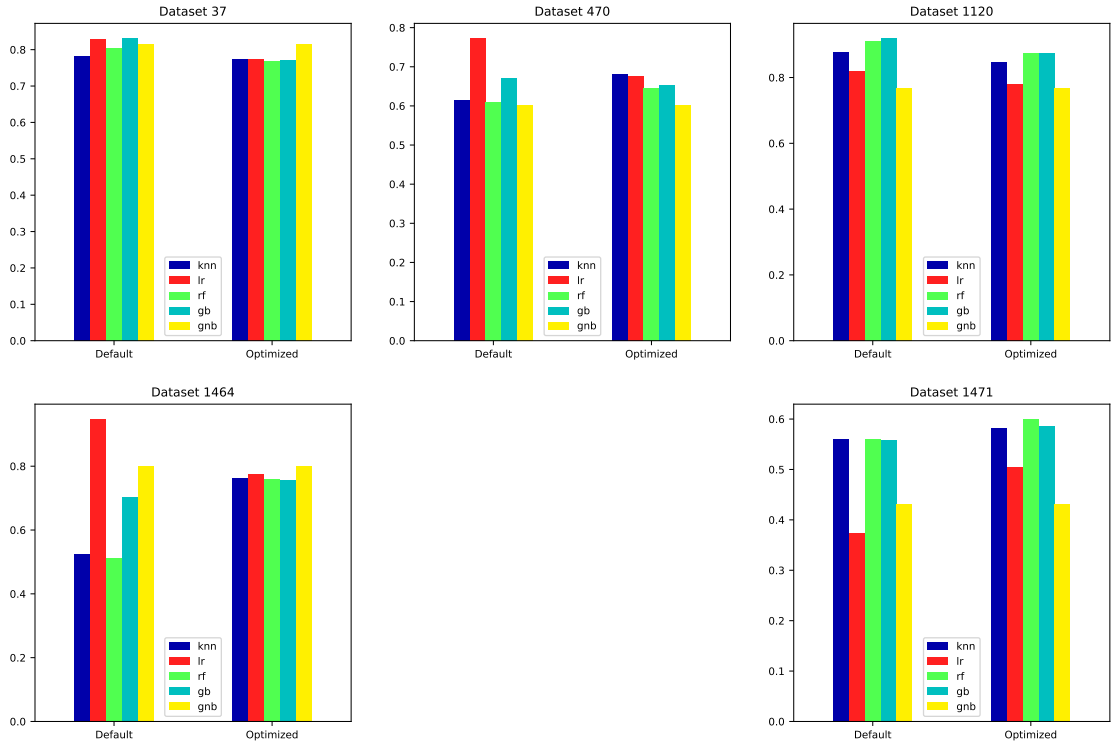
```
Dataset 1120
```

	Default	Optimized
knn	0.88	0.85
lr	0.82	0.78
rf	0.91	0.87
gb	0.92	0.87
gnb	0.77	0.77

```
Dataset 1464
```

	Default	Optimized
knn	0.53	0.76
lr	0.95	0.77
rf	0.51	0.76
gb	0.70	0.76
gnb	0.80	0.80

```
Dataset 1471

     Default  Optimized
knn     0.56       0.58
lr      0.37       0.51
rf      0.56       0.60
gb      0.56       0.59
gnb     0.43       0.43
```



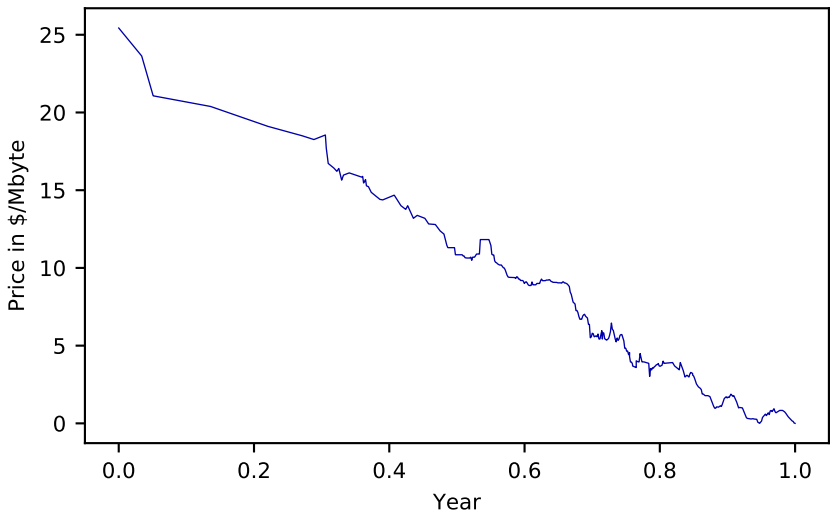
The optimized performances are often similar to those obtained with the default parameters, but in some cases they are significantly lower or higher. This is suspicious, and could stem either from an implementation error, or simply indicate substantial overfitting, manifested through high variance of the models.

Generally gradient boosting and random forests perform very similarly across all datasets, but it is difficult to find a trend, suggesting that, indeed, there is no silver bullet among these algorithms, even though, in most cases, they differ by "only" 10 to 20%.

1.4 Gaussian Processes (2 points (1+1))

Consider the RAM prices dataset (included in the data folder). Separate the data in a training set of all data points up until the year 2000, and a test set with all points after that.

In [2]:
```
ram_prices = pd.read_csv('data/ram_price.csv')

scaler_x = MinMaxScaler().fit(
    np.array(ram_prices.date).reshape((len(ram_prices), 1))
)

ymin = np.log(np.array(ram_prices.price)).min()

def split(cond, scale=True):
    df = ram_prices[cond] if cond is not None else ram_prices
    X = np.array(df.date).reshape((len(df), 1))
    y = np.array(df.price)

    if scale:
        return scaler_x.transform(X), np.log(y) - ymin

    return X, y


def split_test(cond, scale=True):
    train = split(cond, scale)
    test = split(~cond, scale) if cond is not None else ([], [])
    return train, test


Xtrain, ytrain = split(ram_prices.date <= 2000)
Xtest, ytest = split(ram_prices.date > 2000)
Xscal, yscal = split(None)

plt.plot(Xscal, yscal);
plt.xlabel("Year");
plt.ylabel("Price in $/Mbyte");
```

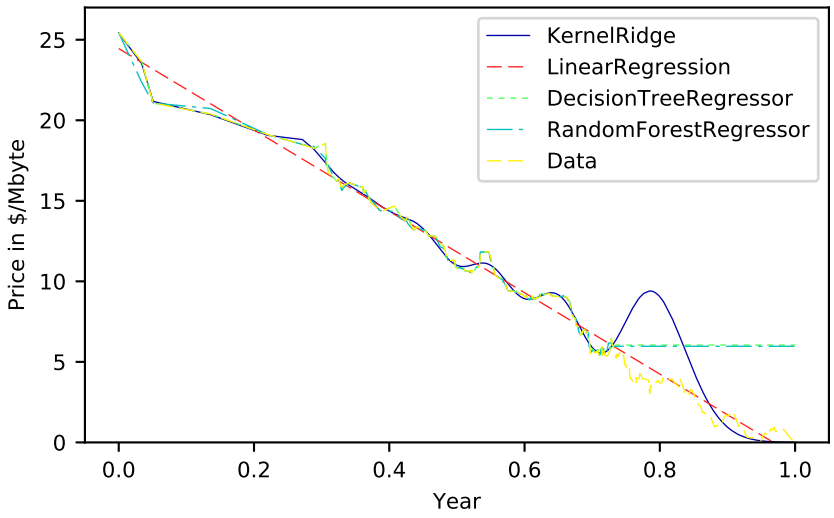


- Train several of the algorithms we have covered in the course that can handle regression. Include at least linear regression, decision tree, and RandomForest. Which ones give the best R^2 performance on the test set? Plot the predictions (both on the training and test data) on the figure below. Use different colors for different algorithms or build multiple plots.

```
In [69]: def fit_eval(rs):
             rs.fit(Xtrain, ytrain)
             r2 = r2_score(ytest, rs.predict(Xtest))
             return rs.best_estimator_, r2


         def train_linear_regression():
             lr = LinearRegression();
             lr.fit(Xtrain, ytrain);
             return lr, r2_score(ytest, lr.predict(Xtest))


         def train_decision_tree():
             grid = {
                 'min_samples_split': [2**x for x in range(1, 6)],
                 'min_samples_leaf': [2**x for x in range(0, 6)],
             }
             return fit_eval(RandomizedSearchCV(
                 DecisionTreeRegressor(), cv=10, n_jobs=-1,
                 n_iter=10, param_distributions=grid
             ))


         def train_random_forest():
             grid = {
                 'n_estimators': [2**x for x in range(0, 6)],
             }
             return fit_eval(GridSearchCV(
                 RandomForestRegressor(), cv=5,
                 n_jobs=-1, param_grid=grid
             ))


         def train_kernel_ridge():
             alphas = [2**x for x in range(-15, 15)]
             grid = [{
                 'kernel': ['linear'],
                 'alpha': alphas
             }, {
                 'kernel': ['rbf'],
```

```
            'alpha': alphas,
            'gamma': alphas
        }, {
            'kernel': ['poly'],
            'alpha': alphas,
            'degree': list(range(2, 11))
        }]
        return fit_eval(GridSearchCV(
            KernelRidge(), cv=5, n_jobs=-1,
            param_grid=grid
        ))


    estimators = [
        train_kernel_ridge(),
        train_linear_regression(),
        train_decision_tree(),
        train_random_forest(),
    ]

    for est, r2 in estimators:
        X, y = split(None)
        plt.plot(X, est.predict(X), label=est.__class__.__name__);
        print('R^2 of %s on test set: %.3g' % (
            est.__class__.__name__, r2
        ));

    plt.plot(Xscal, yscal, label='Data')
    plt.legend(loc='auto')
    plt.xlabel('Year');
    plt.ylabel('Price in $/Mbyte');
    plt.ylim((0, 27))
    plt.show();
```

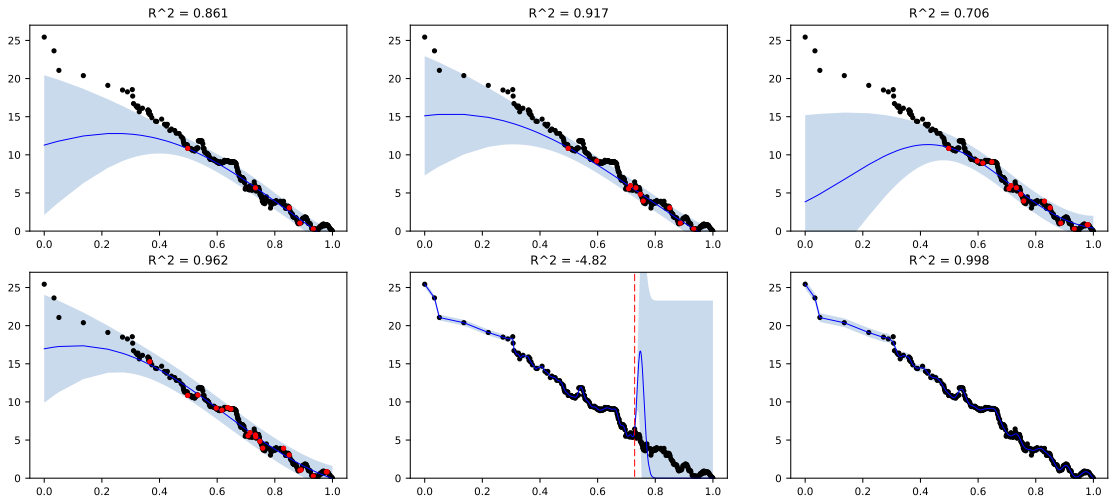
```
R^2 of KernelRidge on test set: -1.44
R^2 of LinearRegression on test set: 0.825
R^2 of DecisionTreeRegressor on test set: -4.39
R^2 of RandomForestRegressor on test set: -4.2
```



- Train a Gaussian process on an increasing amount of samples of the training data. Start with 5 random sample and plot the predictions (both the mean and the uncertainty interval) for both training and test data, as shown in class. Now add 5 more points and retrain and redraw. Do this a couple of times and interpret/explain what you see. Finally, train the Gaussian on the full dataset and again show plot the predictions. Evaluate on the test set using R^2. Compare these results with those achieved with other algorithms and explain.

```
In [6]: rand = np.random.randint(len(ram_prices), size=20)
        sets = [
            split_test(ram_prices.index.isin(rand[:5])),
            split_test(ram_prices.index.isin(rand[:10])),
            split_test(ram_prices.index.isin(rand[:15])),
            split_test(ram_prices.index.isin(rand[:20])),
            split_test(ram_prices.date < 2000),
            split_test(None),
        ]

        Xtest, ytest = split(ram_prices.date >= 2000)

        fig = plt.figure(figsize=(15, 10))
        for i, ((Xtr, ytr), (Xtst, ytst)) in enumerate(sets):
            kernel = WhiteKernel(
                1.0, (1e-3, 1e1)
            ) + ConstantKernel(
                1.0, (1e-5, 1e5)
            ) * RBF(10.0, (1e-5, 1e5))

            gp = GaussianProcessRegressor(
                kernel=kernel, n_restarts_optimizer=10
```

```
        )
        gp.fit(Xtr, ytr);
        y_pred, sigma = gp.predict(Xscal, return_std=True)

        plt.subplot(3, 3, i + 1);
        plt.plot(Xscal, yscal, 'k.');
        plt.plot(Xscal, y_pred, 'b-');
        plt.fill(np.concatenate([Xscal, Xscal[::-1]]),
                 np.concatenate([y_pred - 1.9600 * sigma,
                                 (y_pred + 1.9600 * sigma)[::-1]]),
                 alpha=.5, fc='#95b5db', ec='None');
        plt.ylim((0, 27))

        if i == 4:
            plt.plot([Xtr[-1], Xtr[-1]], [-1e10, 1e10], 'r--');
        elif i < 4:
            plt.plot(Xtr, ytr, 'r.');

        if len(Xtst) > 0:
            r2 = r2_score(ytst, gp.predict(Xtst));
        else:
            r2 = r2_score(ytr, gp.predict(Xtr));

        plt.title('R^2 = %.3g' % r2)

    clear_output();
```



It is important to notice that not applying the log transformation to the price lead to terrible models with very poor generalization accuracy, even between adjacent training samples (maybe a different kernel would help).

It is possible to obtain very good R^2 fits even with few training points, especially if they are

spread uniformly, although the confidence of the prediction tends to get higher as the training data increases. This leads the GP to be overconfident when trained with all the data points before year 2000, and the effect of the little bump at the end of that period is amplificated by the model, which then plummets to zero (GPs naturally return to the mean value when there is no training data nearby, which is the reason why the price was scaled so that the minimum is zero). This behaviour is similar to the one presented by kernel ridge regression, which was a little smoother on the training data.

The plot shown in the previous point shows the inadeguacy of decision trees and random forests for this kind of task: even though they are good at interpolating between the training samples, their extrapolation capabilities are non-existent, and they completely fail to capture the general decreasing trend of the price.

After applying the log transformation, the best R^2 score is achieved by the humble linear regressor. The reason for this is simple: the data does present a linear trend, but the inherent noise tricks more sophisticated models.

1.4.1 A mini-data mining challenge (2 points (+1))

The goal here is to use everything you have learned to build the best model for a given classification task. The task is hosted on OpenML, so you will receive the train-test splits, and your model will be evaluated on the server. The goal is to reasonably select algorithms and hyperparameter settings to obtain the best model. You can also do model selection and parameter optimization as you have done before. Skeleton code is provided in the OpenML tutorial.

- All details can be found online:
  - The OpenML Task ID is 145677: https://www.openml.org/t/145677
  - The dataset description can be found here: https://www.openml.org/d/4134
- A leaderboard is kept of the best models: https://www.openml.org/t/145677#!people
  - You are able to see the solutions of others (by clicking in the timeline or run list), but resubmission of the exact same solution does not register on the leaderboard.
  - You can share one account (one API key) per team. In case you use two, we take the one that performs best.
- You can document the different experiments that you ran in this notebook. For each experiment, provide a description of how you chose the algorithms and parameters that you submitted. Try to reason about which experiments to try, don't just do an immense random search.
- Points are rewarded as follows:
  - 1 point for the breadth of experiments you ran (algorithms, hyperparameter settings)
  - 1 point for reasoning/insight and interpretation of the results
  - 1 (bonus) point for every team who has uploaded the best solution thus far **on AUC** (who reaches the top of the leaderboard at any moment during the assignment)
    - Note: On the leaderboard page, the 'frontier' line is drawn, and your top ranking is also shown in the table.

Note: Report AUC scores in your report as well. In case of issues with OpenML we will use the experiments and scores mentioned your report.

```
In [3]: task = tasks.get_task(145677)

        svd_params = {
            'red': [TruncatedSVD()],
            'red__n_components': [10, 50, 100],
        }

        kbest_params = {
            'red': [SelectKBest()],
            'red__score_func': [f_regression],
            'red__k': [10, 50, 100],
        }

        mod_params = {
            'red': [SelectFromModel(RandomForestClassifier())],
            'red__estimator__max_features': [0.1],
            'red__estimator__n_estimators': [64],
        }


        def test(estimator, grid, n_iter=10, feature_selection=True):
            X, y = shuffle(*task.get_X_and_y())

            est_params = {'est__' + k: v for k, v in grid.items()}

            search_est = Pipeline([
                ('red', None),
                ('est', estimator)
            ]) if feature_selection else estimator

            search_grid = [
                dict(**est_params),
                dict(**est_params, **svd_params),
                dict(**est_params, **kbest_params),
                dict(**est_params, **mod_params),
            ] if feature_selection else grid

            gs = GridSearchCV(search_est, param_grid=search_grid, cv=5,
                              n_jobs=-1, scoring='roc_auc', verbose=3);
            gs.fit(X, y);
            clear_output()

            return gs.best_score_, gs.best_params_
```

If feature selection is enabled, every method is tested with three different strategies, namely dimensionality reduction through truncated SVD, selecting the features that maximize the explained variance (through ANOVA), and selection through random forests, whose parameters were determined by balancing good predictive performances (with max_features=0.1, based on the test run

below) and running performances (by using only 64 estimators). Scaling is not necessary, as the features are already scaled.

```
In [4]: best = test(RandomForestClassifier(), {
            'n_estimators': [2**x for x in range(4, 10, 2)],
            'max_features': np.arange(0.1, 0.6, 0.1),
            'min_samples_split': [2**x for x in range(1, 5)],
            'min_samples_leaf': [2**x for x in range(1, 5)],
        }, feature_selection=False)
        print('rf best score: {:.3f} obtained with params {}'.format(*best))

rf best score: 0.879 obtained with params {'max_features': 0.10000000000000001, 'mi
```

Random forests is one of the first go-to methods when approaching a dataset, as they do not require feature engineering and are pretty robust to overfitting. In fact, no particular tuning was necessary to achieve good performances. Clearly, feature selection methods were disabled in this case, even though it could make sense to apply the truncated SVD.

```
In [5]: best = test(LogisticRegression(), {
            'C':  [2**x for x in range(-24, 12, 3)],
        })

        print('lr best score: {:.3f} obtained with params {}'.format(*best))

lr best score: 0.835 obtained with params {'est__C': 0.015625}
```

Logistic regression is another easy method to try. Its performances are worse than random forest, suggesting that the classes are not linearly well separable. This justifies turning the attention to non-linear models instead.

```
In [7]: best = test(KNeighborsClassifier(), {
            'n_neighbors': [1, 3, 5, 7, 10, 25],
        })

        print('knn best score: {:.3f} obtained with params {}'.format(*best))

knn best score: 0.837 obtained with params {'est__n_neighbors': 5, 'red': SelectFro
            max_depth=None, max_features=0.1, max_leaf_nodes=None,
            min_impurity_split=1e-07, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            n_estimators=64, n_jobs=1, oob_score=False, random_state=None,
            verbose=0, warm_start=False),
        prefit=False, threshold=None), 'red__estimator__max_features': 0.1, 'red__e
```

K-nearest neighbors is first due to its simplicity. The number of neighbors to try is low, as high values would only cause underfit, but too low values cause overfit. Usually a good number of

neighbors is around 5, which is the reason why the grid tests many values around it. Eventually, though, knn achieves performances comparable to those of logistic regression with some help by the random forest selector, which is a bit surprising given that knn suffers in high dimensional spaces due to the curse of dimensionality.

```
In [9]: best = test(SVC(), {
            'kernel': ['rbf'],
            'C': [2**x for x in range(-15, 15, 3)],
            'gamma': [2**x for x in range(-15, 15, 3)],
        })

        print('svc best score: {:.3f} obtained with params {}'.format(*best))
```

```
svc best score: 0.864 obtained with params {'est__C': 1, 'est__gamma': 0.125, 'est_
            max_depth=None, max_features=0.1, max_leaf_nodes=None,
            min_impurity_split=1e-07, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            n_estimators=64, n_jobs=1, oob_score=False, random_state=None,
            verbose=0, warm_start=False),
        prefit=False, threshold=None), 'red__estimator__max_features': 0.1, 'red__e
```

SVM with RBF kernel and neutral regularization parameters achieve performances close to random forests', even though they need help in selecting relevant features.

Overall, random forests are the best predictive model, as well as the best feature selection method. This suggests that the grid used for feature selection through ANOVA was wrong; since feature selection through random forests use the mean feature importance as threshold, it seems like the 100 most correlated features are still not enough, and values around 500/750 should be tried instead. SVD is an unsupervised method, so it does not guarantee a good class separation in the projected space (which is apparently what happened in this case), but scikit's implementation of LDA is for a multiclass setting, thus it is not applicable here.

This situation could be similar, in principle, to what happened in the exercise gaussian processes: complex models tended to overfit the noise in the data, resulting in lower performances. In this case, this means that the separation between classes is noisy and not very defined, similar to the one shown in the first exercise about kernel selection; even there, the simplest explanation won.

```
In [74]: final = RandomForestClassifier(
             max_features=0.1, min_samples_leaf=2, min_samples_split=4,
             n_estimators=256, n_jobs=-1
         )

         #run = runs.run_task(task, final)
         #run.publish()
         #print("Uploaded run with id %s. Check it at www.openml.org/r/%s" %(run.ru
```