Madelon Report

November 10, 2017

In [1]:
```
%run data_package_loading.py # Code loads data as well as packages that are relevant acr
%matplotlib inline


# !conda install -y psycopg2

from sklearn.feature_selection import SelectKBest, RFE, SelectFromModel, RFECV
from sklearn.decomposition import PCA

from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC

from sklearn.model_selection import train_test_split
from sklearn.grid_search import GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline


from sklearn.ensemble import RandomForestClassifier
from tqdm import tqdm

Xdb_1 = pd.read_pickle('data/madelon_db_1')
Xdb_2 = pd.read_pickle('data/madelon_db_2')
Xdb_3 = pd.read_pickle('data/madelon_db_3')


ydb_1 = Xdb_1['target']
ydb_2 = Xdb_2['target']
ydb_3 = Xdb_3['target']
Xdb_1 = Xdb_1.drop(['_id', 'target'], axis=1)
Xdb_2 = Xdb_2.drop(['_id', 'target'], axis=1)
Xdb_3 = Xdb_3.drop(['_id', 'target'], axis=1)

from sklearn.metrics import roc_auc_score, accuracy_score
from sklearn.preprocessing import MinMaxScaler
```

```
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier
```

```
/opt/conda/lib/python3.6/site-packages/sklearn/cross_validation.py:44: DeprecationWarning: This
  "This module will be removed in 0.20.", DeprecationWarning)
/opt/conda/lib/python3.6/site-packages/sklearn/grid_search.py:43: DeprecationWarning: This modul
  DeprecationWarning)
```

0.1 Project 3 - Madelon

Daniel Johnston

DSI-plus2 SEA Github Repository
Problem Statement
The Madelon data as described by UCI: MADELON is an artificial dataset, which was part of the NIPS 2003 feature selection challenge. This is a two-class classification problem with continuous input variables. The difficulty is that the problem is multivariate and highly non-linear.

The objective is to develop a series of models for two purposes:

1. Identifying relevant features.
2. Generating predictions from the model.
   - Models will be scored on Accuracy, as this is a conventional metric for classification problem.

Agenda: 1. Exploratory Data Analysis 1. Benchmarking 1. Feature Selection 1. Secondary EDA 1. Model Pipeline Development 1. Final Model Execution

0.2 1. EDA

We have 6 different datasets for this project. * 3 samples of the UCI sourced data, each with 440 rows and 500 features. These are labeled `uci_1`, `uci_2`, and `uci_3` * 3 samples of the database sourced data, each with ~20000 rows and 1000 features. These are labeled `db_1`, `db_2`, and `db_3` * Sample size varies based on the `TABLESAMPLE` arguement in `postgresql`

Let's take a look at the data 1. Confirm the shape 2. Distribution of target 3. A sample of feature distributions 4. A sample of correlations between features and target

The complete output of charts and relevant code can be found in 0_EDA.ipynb

0.2.1 1-1. Confirm the shape

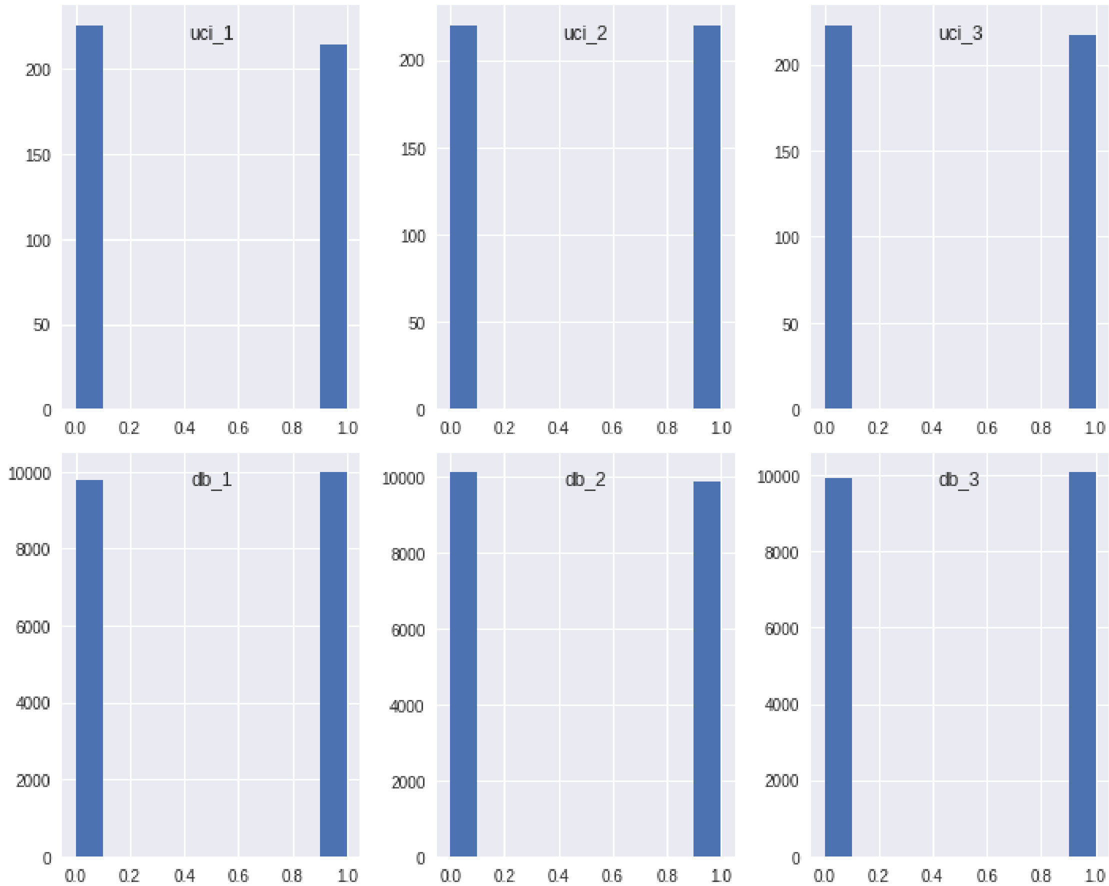
```
In [2]: for Xy in [(Xuci_1, yuci_1, 'uci_1'), (Xuci_2, yuci_2, 'uci_2'),
                   (Xuci_3, yuci_3, 'uci_3'), (Xdb_1,  ydb_1, 'db_1'),
                   (Xdb_2,  ydb_2, 'db_2'), (Xdb_3,  ydb_3, 'db_3')]:
            print("\nOverview of", Xy[2])
            print("X shape:", Xy[0].shape)
            print("y shape:", Xy[1].shape)
```

```
Overview of uci_1
X shape: (440, 500)
y shape: (440,)

Overview of uci_2
X shape: (440, 500)
y shape: (440,)

Overview of uci_3
X shape: (440, 500)
y shape: (440,)

Overview of db_1
X shape: (19791, 1000)
y shape: (19791,)

Overview of db_2
X shape: (20006, 1000)
y shape: (20006,)

Overview of db_3
X shape: (20010, 1000)
y shape: (20010,)
```

0.2.2 1-2. Confirm the distribution of the target

```
In [3]: fig = plt.figure(figsize=(10,8))

        for i, y in enumerate([(yuci_1, 'uci_1'), (yuci_2, 'uci_2'),
                               (yuci_3, 'uci_3'), ( ydb_1, 'db_1'),
                               ( ydb_2, 'db_2'), ( ydb_3, 'db_3')]):
            fig.add_subplot(2,3,1+i)
            plt.hist(y[0])
            plt.title(y[1], y=0.90)


        plt.tight_layout()
        print("It appears that the target classes are equally distributed.")
```

```
It appears that the target classes are equally distributed.
```

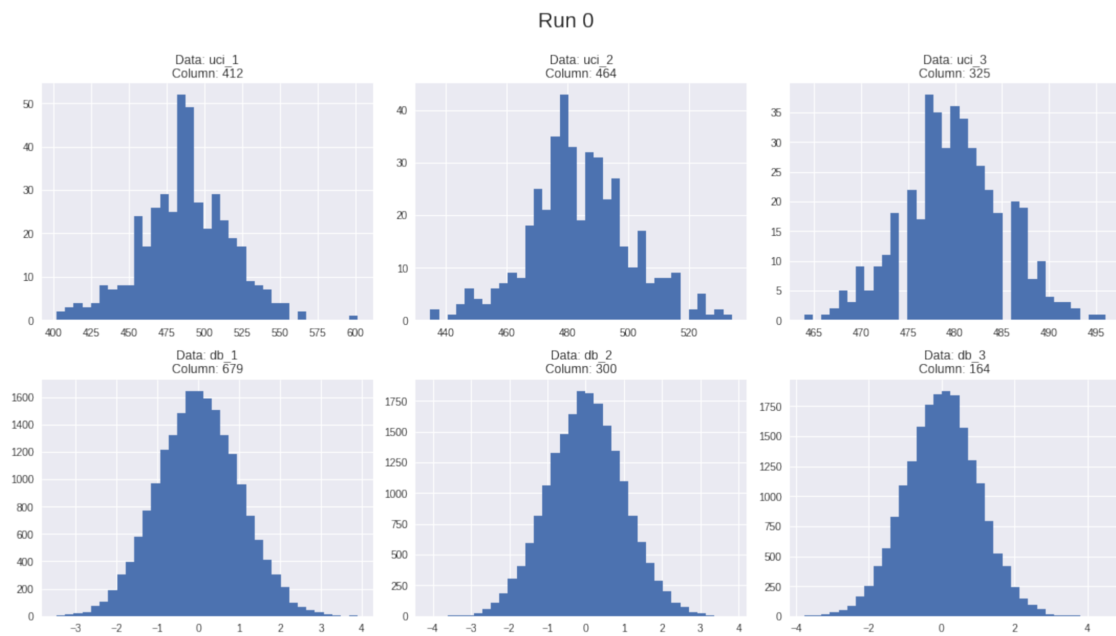


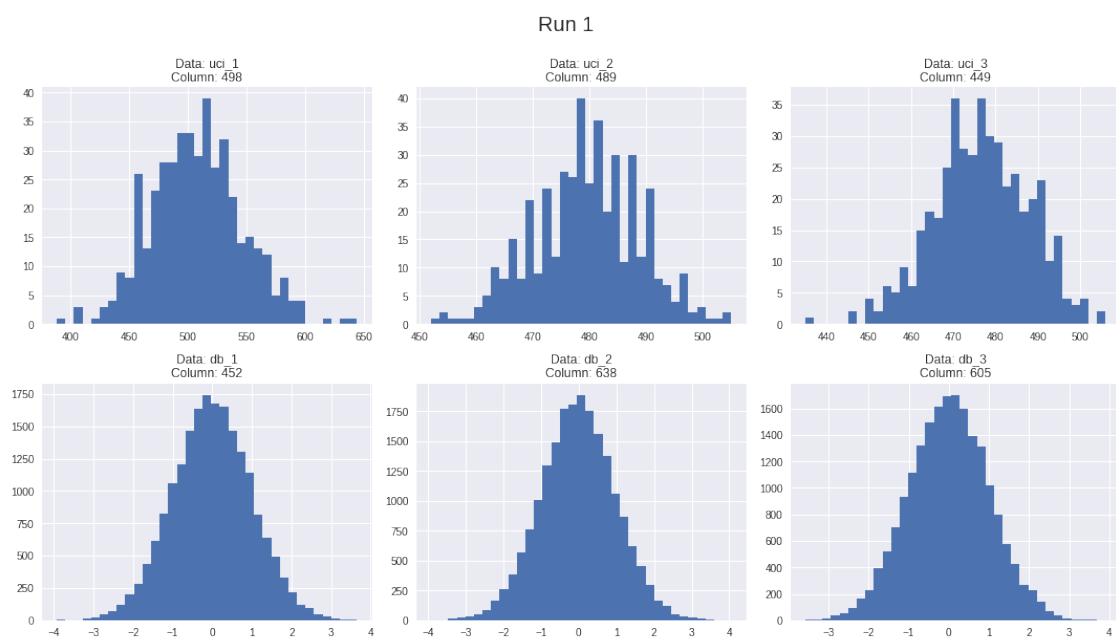
0.2.3 1-3. A sample of feature distributions

Due to the number of features, plotting graphs for all features would be of limited value. At first glance, features appear to roughly normal. Histograms based on UCI data are more noisy due to the limited number of cases within each sample.

```
In [4]: data_target = [(Xuci_1, yuci_1, 'uci_1'),
                       (Xuci_2, yuci_2, 'uci_2'),
                       (Xuci_3, yuci_3, 'uci_3'),
                       (Xdb_1,  ydb_1, 'db_1'),
                       (Xdb_2,  ydb_2, 'db_2'),
                       (Xdb_3,  ydb_3, 'db_3')]


        for run in range(2):
            fig = plt.figure(figsize=(15,8))
            fig.suptitle("Run {}".format(run), fontsize=20, y=1.05)
            for i in range(len(data_target)):
                n_cols = len(data_target[i][0].columns)
                col_i = np.random.randint(0, n_cols)
```

```
        temp_data = data_target[i][0][[col_i]]

        fig.add_subplot(2,3,i+1)
        plt.hist(temp_data.iloc[:, 0], bins = 35)
        plt.title("Data: " + data_target[i][2] + "\nColumn: " + str(col_i))

    plt.tight_layout()
```

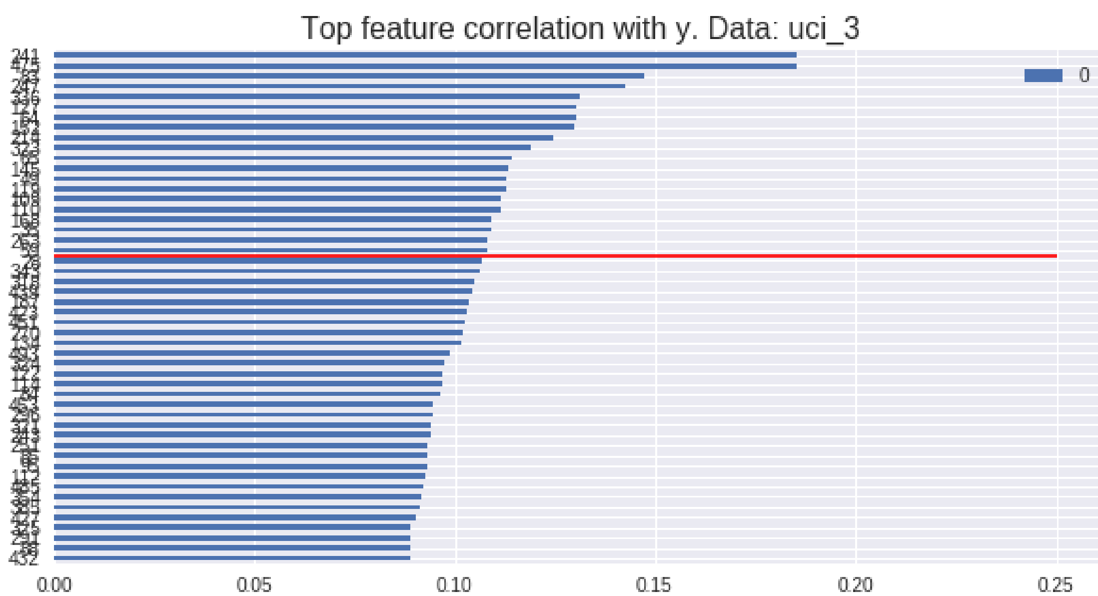


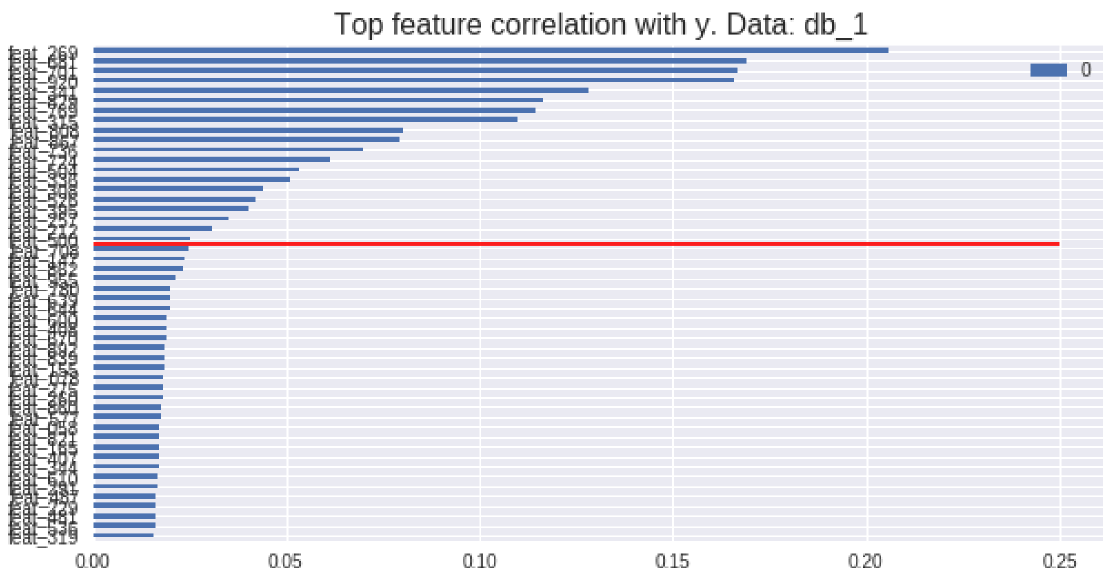


0.2.4 1-4. A sample of correlations between features and target

Please note that not all datasets are shown here. There appear to be some features that are clearly more correlated with the target than others. However, we are expecting 20 informative features in the UCI data and an unknown number in the DB data. These correlations are not clear enough for us to conclusively identify the informative features.

```
In [5]: for Xy in data_target[2:4]:
            X = Xy[0]
            y = Xy[1]

            temp_corr = pd.DataFrame(abs(X.corrwith(y))) # using absoluted value to look at the
            temp_corr = temp_corr.sort_values(0, ascending=False)

            temp_corr.head(50).plot.barh(figsize=(10, 5)).invert_yaxis()
            plt.hlines(20-0.5, 0, 0.25, colors='red') #cutoff to illustrate top 20 features. May
            plt.title("Top feature correlation with y. Data: " + Xy[2], fontsize = 16)
            plt.show()
```





0.3 2. Benchmarking

In order to help assess the value of our work, it is important to give ourselves some baseline prediction scores. As we saw during our EDA, the target distribution is functionally uniform, i.e. 50/50.

Before we invest too much time in feature selection and engineering, let us test a few different models. These models are naive (using the default settings.) These naive models are generally used to help inform us if our model tuning is helping or hurting. There is also a chance that a naive model will be very successful; from the description of the data as well as our EDA, we doubt that this will be the case.

To avoid overfitting, we did set the regularisation parameter to large value: `C = 10 ** 9`

The complete output and relevant code can be found in 1_Benchmarking.ipynb

```
In [6]: benchmark_results_df = pd.read_csv('data/benchmark_results_df.csv')
        # benchmark_results_df.head()
```

0.3.1 2-1. Logistic Regression

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(C = C)
```

As we can see, the `LogisticRegression` model offeres offers only marginal improvement over randomly guessing a classification.

```
In [7]: benchmark_results_df[benchmark_results_df['model'].str.contains('Log') &
                             benchmark_results_df['test_train'].str.contains('test')]
```

```
Out[7]:      data                                              model  \
        1   uci_1   LogisticRegression(C=1000000000, class_weight=...
```

```
9   uci_2  LogisticRegression(C=1000000000, class_weight=...
17  uci_3  LogisticRegression(C=1000000000, class_weight=...
25   db_1  LogisticRegression(C=1000000000, class_weight=...
33   db_2  LogisticRegression(C=1000000000, class_weight=...
41   db_3  LogisticRegression(C=1000000000, class_weight=...

                                               scaler     score test_train
1   StandardScaler(copy=True, with_mean=True, with...  0.466462       test
9   StandardScaler(copy=True, with_mean=True, with...  0.552273       test
17  StandardScaler(copy=True, with_mean=True, with...  0.522887       test
25  StandardScaler(copy=True, with_mean=True, with...  0.527293       test
33  StandardScaler(copy=True, with_mean=True, with...  0.529960       test
41  StandardScaler(copy=True, with_mean=True, with...  0.531837       test
```

0.3.2 2-2. Decision Tree

```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier()
```

Decision tress are looking better than `LogisticRegression` but still not very useful.

```
In [8]: benchmark_results_df[benchmark_results_df['model'].str.contains('Dec') &
                             benchmark_results_df['test_train'].str.contains('test')]
```

```
Out[8]:      data                                              model  \
        3   uci_1  DecisionTreeClassifier(class_weight=None, crit...
        11  uci_2  DecisionTreeClassifier(class_weight=None, crit...
        19  uci_3  DecisionTreeClassifier(class_weight=None, crit...
        27   db_1  DecisionTreeClassifier(class_weight=None, crit...
        35   db_2  DecisionTreeClassifier(class_weight=None, crit...
        43   db_3  DecisionTreeClassifier(class_weight=None, crit...

                                                       scaler     score test_train
        3   StandardScaler(copy=True, with_mean=True, with...  0.517785       test
        11  StandardScaler(copy=True, with_mean=True, with...  0.597727       test
        19  StandardScaler(copy=True, with_mean=True, with...  0.504547       test
        27  StandardScaler(copy=True, with_mean=True, with...  0.612137       test
        35  StandardScaler(copy=True, with_mean=True, with...  0.600156       test
        43  StandardScaler(copy=True, with_mean=True, with...  0.597360       test
```

0.3.3 2-3. K Nearest Neighbors

```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_jobs=-1)
```

Nearest Neighbors splits the difference between logistic regression and decision trees

```
In [9]: benchmark_results_df[benchmark_results_df['model'].str.contains('KN') &
                             benchmark_results_df['test_train'].str.contains('test')]
```

```
Out[9]:      data                                              model  \
        5   uci_1  KNeighborsClassifier(algorithm='auto', leaf_si...
        13  uci_2  KNeighborsClassifier(algorithm='auto', leaf_si...
        21  uci_3  KNeighborsClassifier(algorithm='auto', leaf_si...
        29   db_1  KNeighborsClassifier(algorithm='auto', leaf_si...
        37   db_2  KNeighborsClassifier(algorithm='auto', leaf_si...
        45   db_3  KNeighborsClassifier(algorithm='auto', leaf_si...

                                                    scaler     score test_train
        5   StandardScaler(copy=True, with_mean=True, with...  0.545793       test
        13  StandardScaler(copy=True, with_mean=True, with...  0.552273       test
        21  StandardScaler(copy=True, with_mean=True, with...  0.544284       test
        29  StandardScaler(copy=True, with_mean=True, with...  0.541417       test
        37  StandardScaler(copy=True, with_mean=True, with...  0.549934       test
        45  StandardScaler(copy=True, with_mean=True, with...  0.528786       test
```

0.3.4 2-4. Support Vector Classification

```
from sklearn.svm import SVC
model = SVC(C = C)
```

The performance of SVC seems to be on par with KNN.

```
In [10]: benchmark_results_df[benchmark_results_df['model'].str.contains('SVC') &
                                    benchmark_results_df['test_train'].str.contains('test')]
```

```
Out[10]:      data                                              model  \
         7   uci_1  SVC(C=1000000000, cache_size=200, class_weight...
         15  uci_2  SVC(C=1000000000, cache_size=200, class_weight...
         23  uci_3  SVC(C=1000000000, cache_size=200, class_weight...
         31   db_1  SVC(C=1000000000, cache_size=200, class_weight...
         39   db_2  SVC(C=1000000000, cache_size=200, class_weight...
         47   db_3  SVC(C=1000000000, cache_size=200, class_weight...

                                                    scaler     score test_train
         7   StandardScaler(copy=True, with_mean=True, with...  0.509298       test
         15  StandardScaler(copy=True, with_mean=True, with...  0.584091       test
         23  StandardScaler(copy=True, with_mean=True, with...  0.581951       test
         31  StandardScaler(copy=True, with_mean=True, with...  0.543935       test
         39  StandardScaler(copy=True, with_mean=True, with...  0.544123       test
         47  StandardScaler(copy=True, with_mean=True, with...  0.562589       test
```

From our benchmarking, we have confirmed our suspision that some feature selection or feature engineering will be needed in order to achieve stronger results.

0.4 3. Feature Selection

Given our EDA and what we know about the data, feature selection will be one of the most steps we take during this project. For the UCI data, we know that there are 20 informative features that we need to identify, with 5 being true predictors and 15 being redundant linear combinations of the 5 true features. We do not know how many informative features there are in the Madelon data, but that it should follow a similar structure of some true predictors and some redundant linear combinations.

This last point is important; we **know** that the informative features are at least partially related to each other. This will be key in identifying the informative features.

Three different techniques were used in trying to identify the informative features with varying levels of success: 1. Target prediction with individual features 2. Feature prediction with other features 3. Feature correlations

The complete notes and relevant code can be found in 2_Feature_Extraction_Iterative_Model_A.ipynb, 2_Feature_Extraction_Iterative_Model_B.ipynb, and 2_Feature_Extraction_Classification_and_Correlation.ipynb

0.4.1 3-1. Target prediction with individual features

Attempts were made to discern feature importance by fitting models of each individual feature against the target. The following naive models were used: * `DecisionTreeClassifier()` * `KNeighborsClassifier()` * `LogisticRegression()`

The results of this approach were inconsistent accross datasets and ultimately unreliable. We expected that this would be the case given the distributions of feature correlations with the target. The results from this approach will not be further discussed.

The following function was used to test the different models:

```
def feature_test(X, y, classifier):
    mean_scores = []

    # Run regresspr with Kfold
    for col in tqdm(X.columns):
        train_scores = []
        test_scores = []

        Xcol = X[[col]]

        # Set up Kfolds split
        skf = StratifiedKFold(n_splits=10, shuffle=True, random_state = 42)
        skf.get_n_splits(Xcol, y)

        for train_cv_index, val_cv_index in skf.split(Xcol, y):
            X_train_temp = Xcol.iloc[train_cv_index, :]
            y_train_temp = y[train_cv_index]
            X_test_temp = Xcol.iloc[val_cv_index, :]
            y_test_temp = y[val_cv_index]

            #instantiate and fit
            model = classifier
            model.fit(X_train_temp, y_train_temp)

            #score
```

```
            train_scores.append(model.score(X_train_temp, y_train_temp))
            test_scores.append(model.score(X_test_temp, y_test_temp))

        #store mean scores for each feature
        mean_scores.append({'feature': col,
                            'train_score': np.array(train_scores).mean(),
                            'test_score': np.array(test_scores).mean()})

    df_scores = pd.DataFrame(mean_scores)
    return df_scores
```

0.4.2 3-2. Feature prediction with other features

As previously mentioned, we **know** that the informative features are largely related to eachother. The original true predictors are independent, but were then used to create linear combinations. This means we can test to see how well features can be predicted by other features to identify those that are most highly related.

DecisionTreeRegressor was used to test the relationship between variables.

The following functions were used:

```
def calculate_r_2_for_feature(data, feature):
    tmp_X = data.drop(feature, axis=1)
    tmp_y = data[feature]

    X_train, X_test, y_train, y_test = train_test_split(tmp_X, tmp_y,test_size=0.25)

    # Pipe to scale and fit
    dtr_pipe = Pipeline([
                         ('scaler', StandardScaler()),
                         ('model', DecisionTreeRegressor())
                         ])

    dtr_pipe.fit(X_train, y_train)

    score = dtr_pipe.score(X_test, y_test)
    return score

def mean_r2_for_feature(data, feature):
    scores = []
    for _ in range(5):
        tmp_score = calculate_r_2_for_feature(data, feature)
        scores.append(tmp_score)

        if tmp_score < 0:
            return np.array(scores).mean()

    scores = np.array(scores)
    return scores.mean()
```

```
X_target = [(Xuci_1, 'uci_1'),
            (Xuci_2, 'uci_2'),
            (Xuci_3, 'uci_3'),
            (Xdb_1, 'db_1'),
            (Xdb_2, 'db_2'),
            (Xdb_3, 'db_3')]

for data_src in X_target:
    results_R2 = []
    data = data_src[0]
    src = data_src[1]

    for feature in tqdm(data.columns):
        results_R2.append([feature, mean_r2_for_feature(data, feature)])

    results_df = pd.DataFrame(results_R2, columns = ['Feature', 'R2'])
    results_df.to_pickle('feature_results_' + src + '.pickle')
```

In [11]:
```
feature_results_uci_1 = pd.read_pickle("feature_results_uci_1.pickle")
feature_results_uci_2 = pd.read_pickle("feature_results_uci_2.pickle")
feature_results_uci_3 = pd.read_pickle("feature_results_uci_3.pickle")

uci_1_related_features = feature_results_uci_1.sort_values('R2', ascending = False).hea
uci_2_related_features = feature_results_uci_2.sort_values('R2', ascending = False).hea
uci_3_related_features = feature_results_uci_3.sort_values('R2', ascending = False).hea

uci_1_related_features = np.array(uci_1_related_features.sort_values())
uci_2_related_features = np.array(uci_2_related_features.sort_values())
uci_3_related_features = np.array(uci_3_related_features.sort_values())
```

In [12]:
```
feature_results_uci_1.sort_values('R2', ascending = False).head(25)
```

Out[12]:

	Feature	R2
64	64	0.958868
336	336	0.956095
451	451	0.953310
28	28	0.952888
128	128	0.950895
318	318	0.948276
281	281	0.945870
433	433	0.943008
105	105	0.941654
453	453	0.940853
472	472	0.940290
48	48	0.938026
475	475	0.937661
153	153	0.936956
378	378	0.935737

```
442        442  0.934861
493        493  0.933505
241        241  0.931829
338        338  0.674691
455        455  0.599619
411        411 -0.810011
52          52 -0.810362
34          34 -0.810780
185        185 -0.824573
429        429 -0.848329
```

As we can see, there is a very noticible drop in R^2 scores after the 20th feature. Scores go from positive to negative!

Let's see if all three datasets returned consistent values:

```
In [13]: print(uci_1_related_features == uci_2_related_features)
         print(uci_1_related_features == uci_3_related_features)
```

```
[ True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True  True  True  True  True]
[ True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True  True  True  True  True]
```

Great success!

What about the database Madelon data? We don't know how many informative features there are. Let's take a look at the resulting R^2 scores.

```
In [14]: feature_results_db_1 = pd.read_pickle("feature_results_db_1.pickle")
         feature_results_db_2 = pd.read_pickle("feature_results_db_2.pickle")
         feature_results_db_3 = pd.read_pickle("feature_results_db_3.pickle")
```

```
In [15]: feature_results_db_1.sort_values('R2', ascending = False).head(25)
```

```
Out[15]:      Feature        R2
         639  feat_639  0.956720
         956  feat_956  0.953808
         269  feat_269  0.908935
         867  feat_867  0.902803
         395  feat_395  0.890928
         341  feat_341  0.890568
         315  feat_315  0.877562
         701  feat_701  0.865676
         736  feat_736  0.854405
         336  feat_336  0.852760
         724  feat_724  0.845802
         920  feat_920  0.832407
         257  feat_257  0.820792
         769  feat_769  0.803359
```

```
308  feat_308  0.798796
829  feat_829  0.797168
504  feat_504  0.781253
808  feat_808  0.776515
526  feat_526  0.757537
681  feat_681  0.739815
535  feat_535 -0.633511
795  feat_795 -0.635518
764  feat_764 -0.687781
452  feat_452 -0.699119
649  feat_649 -0.700631
```

It looks like we are seeing the same drop to negative values we saw in the UCI data around the 20th feature in the DB data. Let's double check this across all three DB samples.

```
In [16]: # Where is the cutoff? How many related features are there?
         for df in [feature_results_db_1, feature_results_db_2, feature_results_db_3]:
             temp_df = df.sort_values('R2', ascending = False)

             counter = 0
             for i in temp_df['R2']:
         #         print(i)
         #         print(counter)
                 if i < 0:
                     print(counter)
                     break
                 counter = counter+1
```

```
20
20
20
```

20 features all around. That is a good sign; let's check if they are all the same features.

```
In [17]: db_1_related_features = feature_results_db_1.sort_values('R2', ascending = False).head(
         db_2_related_features = feature_results_db_2.sort_values('R2', ascending = False).head(
         db_3_related_features = feature_results_db_3.sort_values('R2', ascending = False).head(

         db_1_related_features = np.array(db_1_related_features.sort_values())
         db_2_related_features = np.array(db_2_related_features.sort_values())
         db_3_related_features = np.array(db_3_related_features.sort_values())
```

```
In [18]: db_1_related_features == db_2_related_features
```

```
Out[18]: array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True,  True], dtype=bool)
```

```
In [19]: db_1_related_features == db_3_related_features
```

```
Out[19]: array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True,  True], dtype=bool)
```

Great success again.

In summary: Each sample of UCI data suggests that the same 20 features are related, giving us high confidence that the following features are predictors of the target:

`[28, 48, 64, 105, 128, 153, 241, 281, 318, 336, 338, 378, 433, 442, 451, 453, 455, 472, 475, 493]`

Each sample of Madelon DB data suggests that the same 20 features are related, giving us high confidence that the following features are predictors of the target:

`[257, 269, 308, 315, 336, 341, 395, 504, 526, 639, 681, 701, 724, 736, 769, 808, 829, 867, 920, 956]`

While this approach appears to provide conclusive results, took a considerable amount of time to fit all of the necessary model. The next approach explores a faster alternative.

0.4.3 3-3. Feature correlations

Using the same intuition that the informative features are realated to one another, let's take a look at the correlation matrix.

```
In [20]: corr_test = Xuci_1.corr()
         corr_test.head()
```

```
Out[20]:           0         1         2         3         4         5         6  \
         0  1.000000  0.001863 -0.030589  0.079044 -0.016020  0.045018 -0.022883
         1  0.001863  1.000000 -0.045873  0.112781  0.072174  0.027823 -0.086126
         2 -0.030589 -0.045873  1.000000 -0.012340 -0.050528  0.060894  0.035209
         3  0.079044  0.112781 -0.012340  1.000000  0.002612  0.064137 -0.011659
         4 -0.016020  0.072174 -0.050528  0.002612  1.000000  0.000710 -0.049369

                   7         8         9    ...          490       491       492  \
         0  0.002588  0.031829  0.026756    ...     0.013224  0.034647  0.023064
         1  0.008964 -0.005845 -0.048256    ...     0.007706 -0.101836 -0.027601
         2 -0.050305  0.059404  0.040547    ...     0.012974  0.044949  0.097373
         3 -0.070683  0.033268 -0.004106    ...     0.032361 -0.031768 -0.043497
         4  0.047841 -0.027970 -0.049692    ...    -0.024900  0.002100  0.002794

                 493       494       495       496       497       498       499
         0 -0.056057 -0.012904 -0.010460  0.017305 -0.055814  0.057639 -0.033592
         1  0.006410  0.049305  0.022063  0.048790  0.001745 -0.037004  0.060490
         2 -0.016076  0.002061 -0.014319  0.039001  0.016703  0.015267  0.088202
         3 -0.000271  0.078904  0.012186  0.041357 -0.061805 -0.050831  0.034481
         4  0.042962  0.012581 -0.008397  0.042619  0.015844 -0.021339 -0.067939

         [5 rows x 500 columns]
```

That is is a very large matrix to manually inspect. Let's see if we can get `python` to make our job easier.

```
In [21]: # zero at the diagonal.
         for i in corr_test.columns:
             corr_test.loc[i,i] = 0

         # take the absolute value of correlations. We only care about the magnitude, not the di

         corr_test = abs(corr_test)

         corr_test.max().sort_values(ascending=False)[:25]
```

```
Out[21]: 64     0.992330
         336    0.992330
         451    0.990578
         28     0.990578
         318    0.990541
         153    0.990379
         281    0.990379
         433    0.990082
         105    0.989993
         128    0.989993
         241    0.988937
         475    0.988937
         48     0.988595
         378    0.988595
         493    0.988309
         453    0.988309
         472    0.988133
         442    0.988133
         455    0.725369
         338    0.685807
         486    0.216672
         269    0.216672
         162    0.205203
         389    0.205203
         144    0.203834
         dtype: float64
```

Similar to the R^2 scores, we are seeing a clear drop in the correlations after the 20th feature. Let's see if we get consistent sets of results across the different samples of data.

```
In [22]: def test_corr(df):
             # get the absolute values of correlations
             corr_df = abs(df.corr())

             # zero out the diagonal
             for i in corr_df.columns:
                 corr_df.loc[i,i] = 0

             top_features = corr_df.max().sort_values(ascending=False)[:20].index
```

```
        return np.array(top_features)

    uci_1_features = test_corr(Xuci_1)
    uci_2_features = test_corr(Xuci_2)
    uci_3_features = test_corr(Xuci_3)

    db_1_features = test_corr(Xdb_1)
    db_2_features = test_corr(Xdb_2)
    db_3_features = test_corr(Xdb_3)

    uci_1_features.sort()
    uci_2_features.sort()
    uci_3_features.sort()
    db_1_features.sort()
    db_2_features.sort()
    db_3_features.sort()
```

UCI data:

```
In [23]: print(uci_1_features == uci_2_features)
         print(uci_1_features == uci_3_features)
```

```
[ True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True  True  True  True  True]
[ True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True  True  True  True  True]
```

Database data:

```
In [24]: print(db_1_features == db_2_features)
         print(db_1_features == db_3_features)
```

```
[ True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True  True  True  True  True]
[ True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True  True  True  True  True]
```

0.4.4 3. Feature Selection Conclusion

We found two approaches that deliver the same set of informative features across all samples of our data.

- Feature prediction with other features
- Feature Correlations

These are both successful for the same core reason: the informative features are related to one another. While the **Feature prediction with other features** strikes us as more robust, due the repeated sampling and rigor of the model, the **Feature Correlations** strikes us as more scalable and efficient.

0.5 4. Secondary EDA

Now that we have identified the 20 informative features for both sets of data, let's take a detour to reinspect the data now that the scale is managable.

The complete output of charts and relevant code can be found in 3_Feature_Importance_EDA_again.ipynb and 3_Feature_Importance_Reduction.ipynb

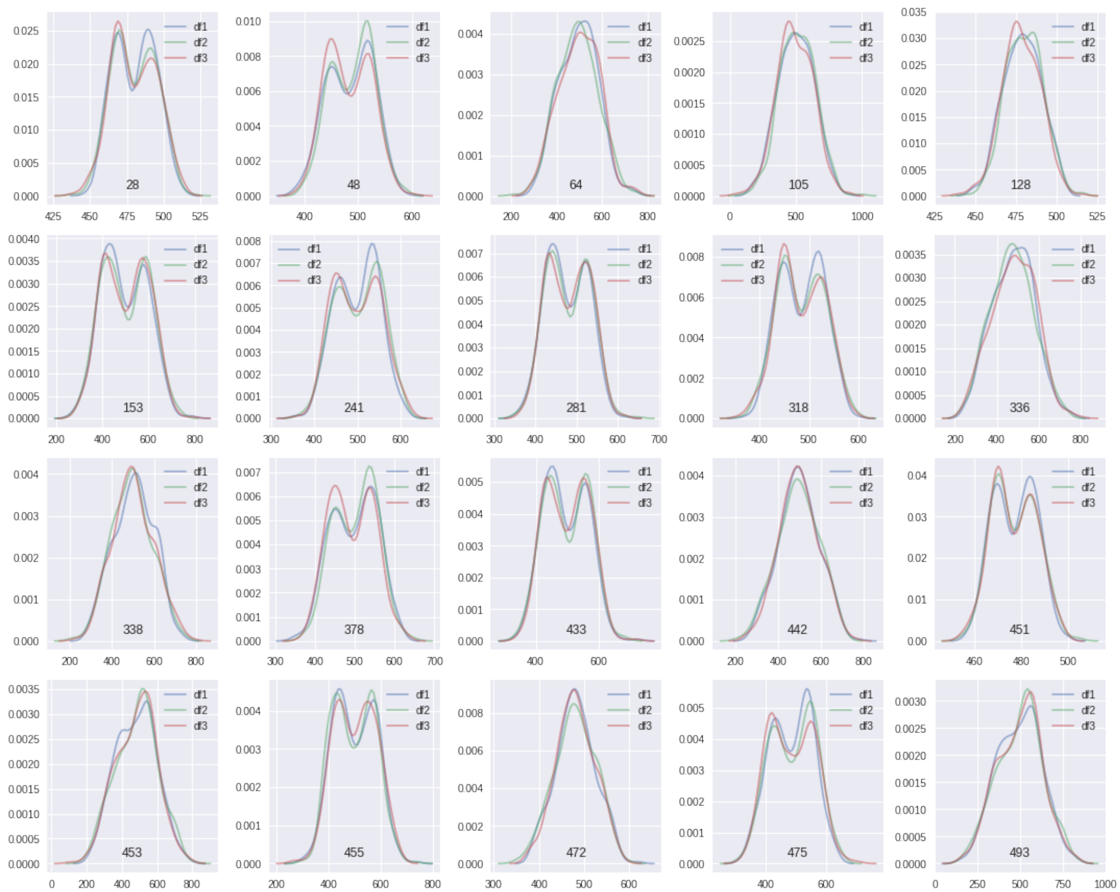
```
In [25]: uci_features = ['28',  '48',  '64', '105', '128', '153', '241', '281', '318', '336',
                         '338', '378', '433', '442', '451', '453', '455', '472', '475', '493']

         madelon_features = ['feat_257', 'feat_269', 'feat_308', 'feat_315', 'feat_336',
                             'feat_341', 'feat_395', 'feat_504', 'feat_526', 'feat_639',
                             'feat_681', 'feat_701', 'feat_724', 'feat_736', 'feat_769',
                             'feat_808', 'feat_829', 'feat_867', 'feat_920', 'feat_956']

         Xuci_1 = Xuci_1[uci_features]
         Xuci_2 = Xuci_2[uci_features]
         Xuci_3 = Xuci_3[uci_features]
         Xdb_1 = Xdb_1[madelon_features]
         Xdb_2 = Xdb_2[madelon_features]
         Xdb_3 = Xdb_3[madelon_features]
```

```
In [26]: def overlayed_kde(df1, df2, df3):
             fig = plt.figure(figsize=(15,12))

             for i, col in enumerate(df1.columns):
                 fig.add_subplot(4,5,i+1)
         #         df[col].hist(bins=30)
                 sns.kdeplot(df1[col], label = 'df1', gridsize=50, alpha=0.5, bw = 'silverman')
                 sns.kdeplot(df2[col], label = 'df2', gridsize=50, alpha=0.5, bw = 'silverman')
                 sns.kdeplot(df3[col], label = 'df3', gridsize=50, alpha=0.5, bw = 'silverman')
                 plt.title(col, y=0.05)

             plt.tight_layout()
```

Let's start by looking at some overlaid KDE plots of all three sets of data from each data source.

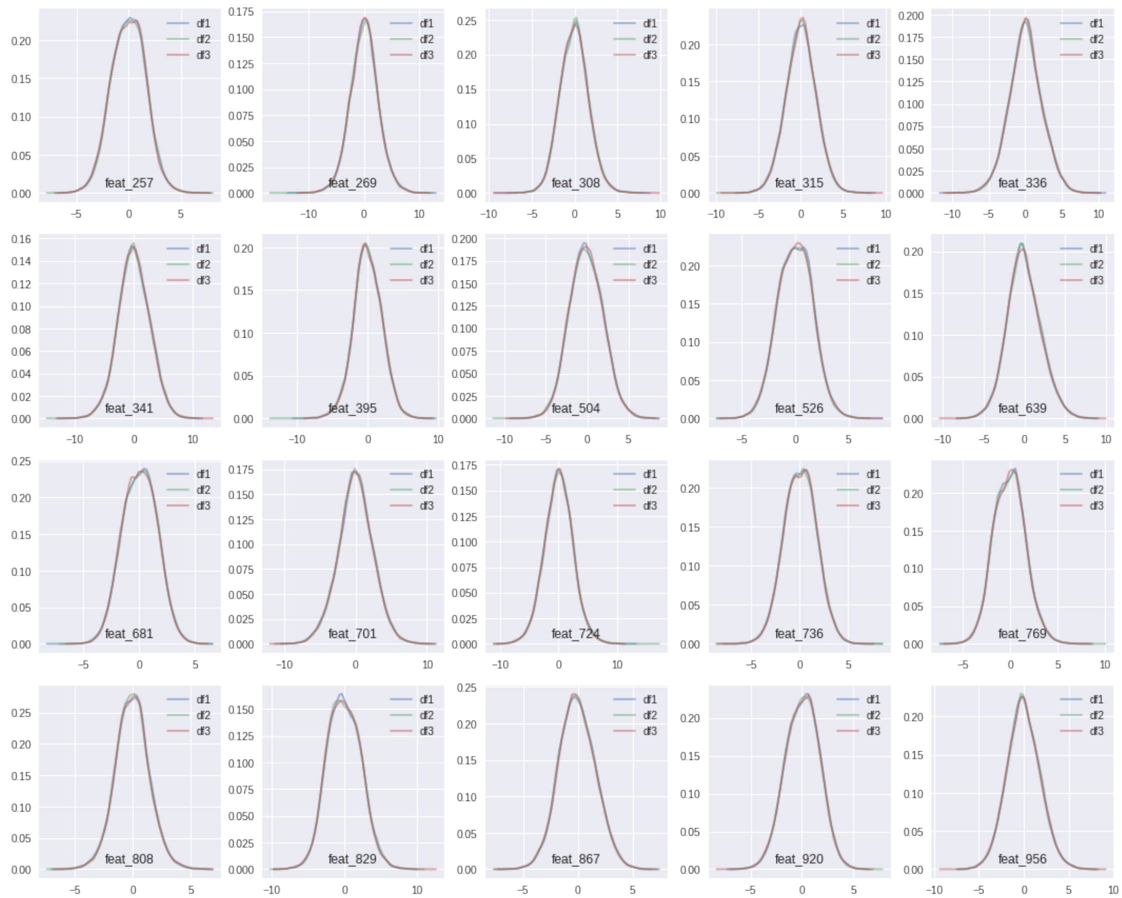
The UCI data:

```
In [27]: overlayed_kde(Xuci_1, Xuci_2, Xuci_3)
```



And the database data:

```
In [28]: overlayed_kde(Xdb_1, Xdb_2, Xdb_3)
```



The UCI data is showing bimodal characteristics in many features. This might be an indication that these features are more informative than the unimodal characteristics. Unfortunately, the same bimodal nature did not appear in the database data, so this avenue of data exploration was discontinued.

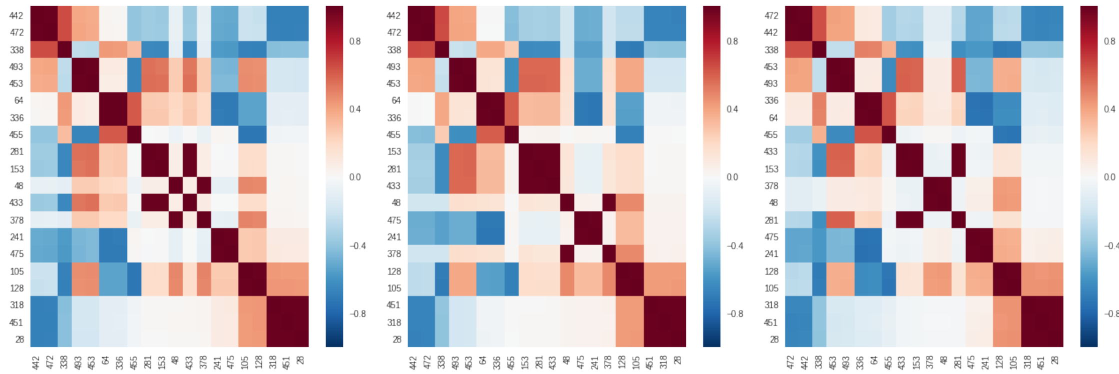
Next, let's compare heatmaps across the different sets of each dataset.

Below are the heatmaps of correlations between features for the UCI data. We have ordered the heatmap such that highly correlated features are placed more closely to one another. We can observe a some very clear patterns in which some features are very highly (0.95 or greater) with eachother. We identified 10 clear groupings.

```
In [29]: def make_heat(df):
             corr_df = df.corr().sort_values(by=df.columns[0]).sort_values(by=df.columns[0],axis
             sns.heatmap(corr_df, cmap='RdBu_r')
```

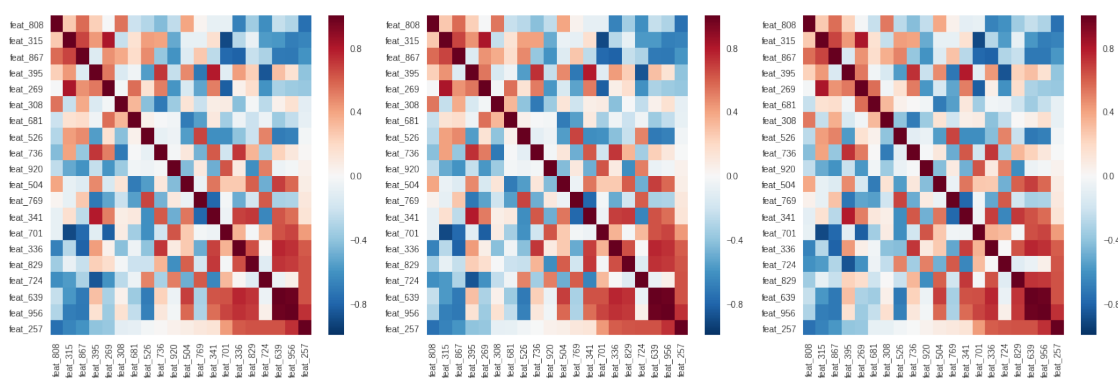
```
In [30]: fig = plt.figure(figsize=(18,6))

         for i, df in enumerate([Xuci_1, Xuci_2, Xuci_3]):
             fig.add_subplot(1,3,1+i)

             make_heat(df)
```

```
plt.tight_layout()
```



Unfortunately, the same can not be said for the DB data. We can visually identify some cases where features seem to be related, but in no instance are there groupings as apparent as with the UCI data.

```
In [31]: fig = plt.figure(figsize=(18,6))

         for i, df in enumerate([Xdb_1, Xdb_2, Xdb_3]):
             fig.add_subplot(1,3,1+i)

             make_heat(df)

         plt.tight_layout()
```



1 5. Model Pipeline Development

The model pipeline development involved the testing and tuning of a wide variety of features selection, dimensionality reduction, and classifier tools.

We ultimately used a `VotingClassifier` ensemble of `RandomForestClassifier`, `KNeighborsClassifier`, and `SVC` with weights of `1.0`, `1.5`, and `0.5` respectively.

Boosting methods such as `AdaBoostClassifier`, `GradientBoostingClassifier`, and `XGBClassifier` were also explored, but ultimately ruled out due to the length of time required to fit these models on large datasets.

The complete output and relevant code can be found in 3_Pipelines.ipynb and 3_Pipelines2.ipynb

1.0.1 Manual grouping of features

We started by trying to manually group features based on a visual inspection of the heatmaps previously show.

This was a dead end, particularly for the Madelon DB data. The correlations weren't as intuitively clear as in the UCI data and the groupings ultimately failed to produce the strongest models.

```
def uci_group(X):
    grouped_df = pd.DataFrame()
    grouped_df['A'] = X[['28','451','318']].mean(axis=1)
    grouped_df['B'] = X[['105','128']].mean(axis=1)
    grouped_df['C'] = X[['241','475']].mean(axis=1)
    grouped_df['D'] = X[['378','48']].mean(axis=1)
    grouped_df['E'] = X[['153','281','433']].mean(axis=1)
    grouped_df['F'] = X[['64','336']].mean(axis=1)
    grouped_df['G'] = X[['453', '493']].mean(axis=1)
    grouped_df['H'] = X[['472','442']].mean(axis=1)
    grouped_df['I'] = X[['338']].mean(axis=1)
    grouped_df['J'] = X[['455']].mean(axis=1)

    return grouped_df

def mad_group(X):
    grouped_df = pd.DataFrame()
    grouped_df['A'] = X[['feat_956','feat_639','feat_829']].mean(axis=1)
    grouped_df['B'] = X[['feat_269','feat_315', 'feat_701']].mean(axis=1) #701 is negative corre
    grouped_df['C'] = X[['feat_341','feat_395']].mean(axis=1)
    grouped_df['D'] = X[['feat_336', 'feat_867']].mean(axis=1)
    grouped_df['E'] = X[['feat_808', 'feat_257']].mean(axis=1)
    grouped_df['F'] = X[['feat_308', 'feat_736']].mean(axis=1)
    grouped_df['G'] = X[['feat_504', 'feat_681']].mean(axis=1)
    grouped_df['H'] = X[['feat_724', 'feat_769']].mean(axis=1)
    grouped_df['I'] = X[['feat_526']].mean(axis=1)
    grouped_df['J'] = X[['feat_920']].mean(axis=1)

    return grouped_df
```

The Grouped datas were used with a variety of different kinds of models, each of which was grid searched to find the optimal parameters.

All model pipelines consist of StandardScaler, SelectKBest, PCA, and a classifier. We also tested pipelines without SKB at this stage, but did not find the results fruitful, so they will be omited from the discussion.

DecisionTree Classifier

- UCI samples' accuraciess were 0.636364, 0.566667, 0.621212
- Madelon DB samples accuracies were 0.613468, 0.615984, 0.625168

```
dtc_pipe = Pipeline([('scaler', StandardScaler()),
                     ('rfe', SelectKBest()),
                     ('pca', PCA()),
                     ('classifier', DecisionTreeClassifier())])

dtc_params = {'rfe__k': [5, 10],
             'pca__n_components': [1, 2, 3, 4, 5],
             'classifier__max_depth': [1, 3, 5, 10, 15, None],
             'classifier__splitter': ['random', 'best']}
```

LogisticRegression Classifier

- UCI samples' accuraciess were 0.542424, 0.606061, 0.569697
- Madelon DB samples accuracies were 0.587205, 0.569851, 0.593540

```
lr_pipe = Pipeline([('scaler', StandardScaler()),
                     ('rfe', SelectKBest()),
                     ('pca', PCA()),
                     ('classifier', LogisticRegression())])

lr_params = {'rfe__k': [5, 10],
             'pca__n_components': [1, 2, 3, 4, 5],
             'classifier__penalty': ['l1', 'l2'],
             'classifier__max_iter': [100, 500],
             'classifier__C': np.logspace(-3,3,7)}
```

KNeighbors Classifier

- UCI samples' accuraciess were 0.696970, 0.690909, 0.736364
- Madelon DB samples accuracies were 0.690236, 0.688109, 0.675639

```
knn_pipe = Pipeline([('scaler', StandardScaler()),
                     ('rfe', SelectKBest()),
                     ('pca', PCA()),
                     ('classifier', KNeighborsClassifier())])

knn_params = {'rfe__k': [5, 10],
             'pca__n_components': [1, 2, 3, 4, 5],
             'classifier__n_neighbors': [1, 5, 9, 15, 25]}
```

RandomForest Classifier

- UCI samples' accuraciess were 0.621212, 0.654545, 0.709091
- Madelon DB samples accuracies were 0.726599, 0.692658, 0.696501

```
rfc_pipe = Pipeline([('scaler', StandardScaler()),
                     ('rfe', SelectKBest()),
                     ('pca', PCA()),
                     ('classifier', RandomForestClassifier())])

rfc_params = {'rfe__k': [5, 10],
             'pca__n_components': [1, 2, 3, 4, 5],
             'classifier__n_estimators': [10, 50, 100, 200, 500],
             'classifier__max_depth': [1, 5, None]}
```

Support Vector Classifier

- UCI samples' accuraciess were 0.700000, 0.681818, 0.721212
- Madelon DB samples accuracies were 0.707744, 0.666017, 0.696501

```
svc_pipe = Pipeline([('scaler', StandardScaler()),
                     ('rfe', SelectKBest()),
                     ('pca', PCA()),
                     ('classifier', SVC())])

svc_params = {'rfe__k': [5, 10],
             'pca__n_components': [1, 2, 3, 4, 5],
              'classifier__C': np.logspace(-3,3,7)}
```

It is at this point where it was becoming apparent that the manual grouping of features was going to result in functionally limited models. It is also important to note that visual inspection to group features is a fundamentally limited approach that can not scale.

One last observation on these results: it is becoming clear that `RandomForestClassifier`, `KNeighborsClassifier`, and `SVC` are the best performing models. Both `LinearRegression` and `DecistionTree` were tested further along with the other classifiers, but will not be discussed further as they were ultimately not used in the final model.

1.0.2 Using top 20 features

It was at this point that pipeline development started to focus primarily on the Madelon DB data. This shift in focus reflects the reach goal to test a model against the full 200k row dataset, and the time required to train and test models on the larger samples of data.

The immediate challenge facing us at this point was to identify the best way to either reduce features or to reduce the dimensionality of our data.

Following are the different pipelines and parameters tested for `RandomForestClassifier`. The same changes were tested across all classifiers.

Using PCA

```
rfc_pca_pipe = Pipeline([('scaler1', StandardScaler()),
                         ('pca', PCA()),
                         ('scaler2', StandardScaler()),
                         ('classifier', RandomForestClassifier())])

rfc_pca_params = {'pca__n_components': [1, 3, 5],
             'classifier__n_estimators': [10, 50, 100, 200, 500],
              'classifier__max_features': ['log2', 'sqrt', 'auto'],
              'classifier__oob_score': [True, False],
             'classifier__max_depth': [1, 5, None]}
```

Using SKB

```
rfc_skb_pipe = Pipeline([('scaler1', StandardScaler()),
                         ('skb', SelectKBest()),
                         ('scaler2', StandardScaler()),
                         ('classifier', RandomForestClassifier())])

rfc_skb_params = {'skb__k': [5, 10, 15],
             'classifier__n_estimators': [10, 50, 100, 200, 500],
              'classifier__max_features': ['log2', 'sqrt', 'auto'],
              'classifier__oob_score': [True, False],
             'classifier__max_depth': [1, 5, None]}
```

Using SFM The model used for SFM:

```
rfc_for_skb = RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='log2', 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=10, n_jobs=1, oob_score=False, random_state=None,
            verbose=0, warm_start=False)

knn_sfm_pipe = Pipeline([('scaler1', StandardScaler()),
                         ('sfm', SelectFromModel(rfc_for_skb)),
                             ('scaler2', StandardScaler()),
                         ('classifier', KNeighborsClassifier())])

rfc_sfm_params = {             'classifier__n_estimators': [10, 50, 100, 200, 500],
              'classifier__max_features': ['log2', 'sqrt', 'auto'],
              'classifier__oob_score': [True, False],
             'classifier__max_depth': [1, 5, None]}
```

1.0.3 Comparing the pipelines

The pipelines are all starting to converg with accuracies in the low 0.8x range (excluding `DecisionTree` and `LogisticRegression`. The models utilizing PCA had mariginally better scores, so they were chosen as the best.

- PCA
  - Decision Tree: 0.751616814875
  - LogReg: 0.601050929669
  - Random Forest: 0.829830234438
  - KNN: 0.837510105093
  - SVC: 0.831447049313
- SelectKBest
  - Decision Tree: 0.746564268391
  - LogReg: 0.601455133387
  - Random Forest: 0.825383993533
  - KNN: 0.823767178658
  - SVC: 0.820735650768
- SelectFromModel
  - Decision Tree: 0.749797898141
  - LogReg: 0.600848827809
  - Random Forest: 0.828011317704
  - KNN: 0.829021827001
  - SVC: 0.822352465643*

In order to try to drive a slightly higher accuracy, we tested a voting ensemble with `VotingClassifier`. The voting ensemble consisted of the optimal models from `RandomForestClassifier`, `KNeighborsClassifier`, and `SVC` utilizing PCA.

Optimal `RandomForestClassifier` Pipeline

```
rfc_pca_classifier =
[('scaler1', StandardScaler(copy=True, with_mean=True, with_std=True)),
 ('pca',
  PCA(copy=True, iterated_power='auto', n_components=5, random_state=None,
    svd_solver='auto', tol=0.0, whiten=False)),
 ('scaler2', StandardScaler(copy=True, with_mean=True, with_std=True)),
 ('classifier',
  RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='sqrt', 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=500, n_jobs=1, oob_score=False, random_state=None,
            verbose=0, warm_start=False))]
```

Optimal `KNeighborsClassifier` Pipeline

```
knn_pca_classifier =
[('scaler1', StandardScaler(copy=True, with_mean=True, with_std=True)),
 ('pca',
  PCA(copy=True, iterated_power='auto', n_components=5, random_state=None,
    svd_solver='auto', tol=0.0, whiten=False)),
```

```
 ('scaler2', StandardScaler(copy=True, with_mean=True, with_std=True)),
 ('classifier',
  KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
             metric_params=None, n_jobs=1, n_neighbors=8, p=2,
             weights='distance'))]
```

Optimal `SVC` **Pipeline**

```
svc_pca_classifier =
[('scaler1', StandardScaler(copy=True, with_mean=True, with_std=True)),
 ('pca',
  PCA(copy=True, iterated_power='auto', n_components=5, random_state=None,
    svd_solver='auto', tol=0.0, whiten=False)),
 ('classifier', SVC(C=10, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',
    max_iter=-1, probability=True, random_state=None, shrinking=True,
    tol=0.001, verbose=False))]
```

Within the `VotingClassifier`, we tested an array of weights `[ 0.5 , 0.75, 1. , 1.25, 1.5 ]` for each component classifier to determine which mix would results in the best accuracy scores. This is how the weights were determined

```
In [32]: voting_scores_df = pd.read_csv('data/election.csv')
         voting_scores_df.sort_values('test_score', ascending=False).head(10)
```

Out[32]:

	test_score	train_score	weights
68	0.842765	1.0	[1.0, 1.5, 0.5]
43	0.842361	1.0	[0.75, 1.5, 0.5]
92	0.841956	1.0	[1.25, 1.5, 0.5]
88	0.841350	1.0	[1.25, 1.25, 0.5]
98	0.840946	1.0	[1.5, 0.5, 0.75]
63	0.840946	1.0	[1.0, 1.25, 0.5]
74	0.840946	1.0	[1.25, 0.5, 0.75]
38	0.840744	1.0	[0.75, 1.25, 0.5]
78	0.840340	1.0	[1.25, 0.75, 0.5]
117	0.840340	1.0	[1.5, 1.5, 0.5]

Our final `VotingClassifier`:

```
final_model = VotingClassifier(estimators = [
                                ('rfc', rfc_pca_classifier),
                                ('knn', knn_pca_classifier),
                                ('svc', svc_pca_classifier)],
                          voting = 'soft',
                          weights = [1.0, 1.5, 0.5]
                         )
```

2 6. Final Model Execution

The complete output and relevant code can be found in Final_model.ipynb

```
final_model.score(X_test, y_test)
```

0.8652499999999996

```
final_probs = final_model.predict_proba(X_test) roc_auc_score(y_test, [prob[1] for prob in final_probs])
```

0.93950507475272993

Ultimately, this model achieved a strong 0.865 accuracy and a robust AUC score of 0.94. It is possible that stronger scores could be achieved through further tuning of the three individual classifiers, as well as the weights using in the `VotingClassifier`