scratch

March 27, 2016

0.1 Data loading

We'll begin by doing PCA on our train + test datasets to see how separable our classes are.

```
In [1]: %matplotlib inline

        import sys
        import numpy as np
        import pandas as pd
        import zipfile
        from sklearn.decomposition import PCA, KernelPCA
        from sklearn import preprocessing
        import matplotlib.pyplot as plt
        import seaborn as sns

        zf_train = zipfile.ZipFile('../../data/train.csv.zip')
        train = pd.DataFrame.from_csv(zf_train.open('train.csv'))

        zf_test = zipfile.ZipFile('../../data/test.csv.zip')
        test = pd.DataFrame.from_csv(zf_test.open('test.csv'))



        ## Dimensions of train set
        ntrain,dtrain = train.shape

        ## Dimensions of test set
        ntest, dtest = test.shape
```

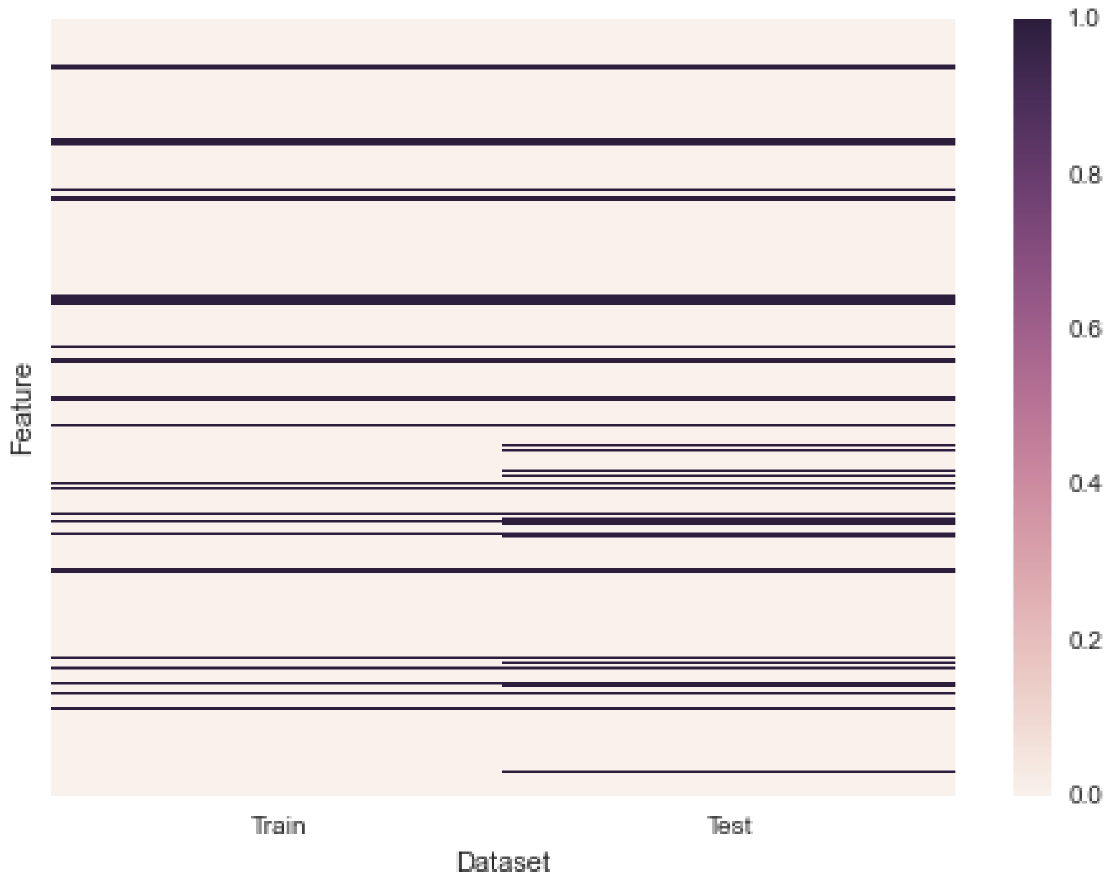
0.2 Columns with zero variance

Columns with zero variance are labeled as 1; columns with variation are labeled as 0.

```
In [2]: a = np.all(train == train.iloc[0,], axis = 0)[0:dtrain-1]
        b = np.all(test == test.iloc[0,], axis = 0)
        identicals = np.column_stack((a,b))

        ax = sns.heatmap(identicals, xticklabels=['Train', 'Test'], yticklabels=False)
        ax.set(xlabel='Dataset', ylabel='Feature')
        plt.show()
```

```
/Users/allen/anaconda/lib/python2.7/site-packages/matplotlib/collections.py:590: FutureWarning: element
  if self._edgecolors == str('face'):
```



This is good news – columns with 0 variance in the test dataset are a subset of those in the train dataset. If this was **not** true, our training set would not be sufficient for classification of our test set.

0.3 Principal components analysis

```
In [2]: def pca_plot(X, targets, title):
            target_names = np.unique(targets)
            pca = PCA(n_components = 2)
            X_r = pca.fit(X).transform(X)
            X_scaled = preprocessing.scale(X_r)
            print('explained variance ratio (first two components): %s'
                  % str(pca.explained_variance_ratio_))

            plt.figure()
            for c, i, target_name in zip("rb", [0, 1], target_names):
                plt.scatter(X_scaled[np.where(targets == i),0], X_scaled[np.where(targets == i),1], c=c
            plt.legend()
            plt.title(title)

        def kernel_pca(X, targets, title, kernel, **kwargs):
            kpca = KernelPCA(kernel=kernel, fit_inverse_transform=True, **kwargs)
            X_kpca = kpca.fit_transform(X)
            X_back = kpca.inverse_transform(X_kpca)
```

```
    pca_plot(X_back, targets, title)
```

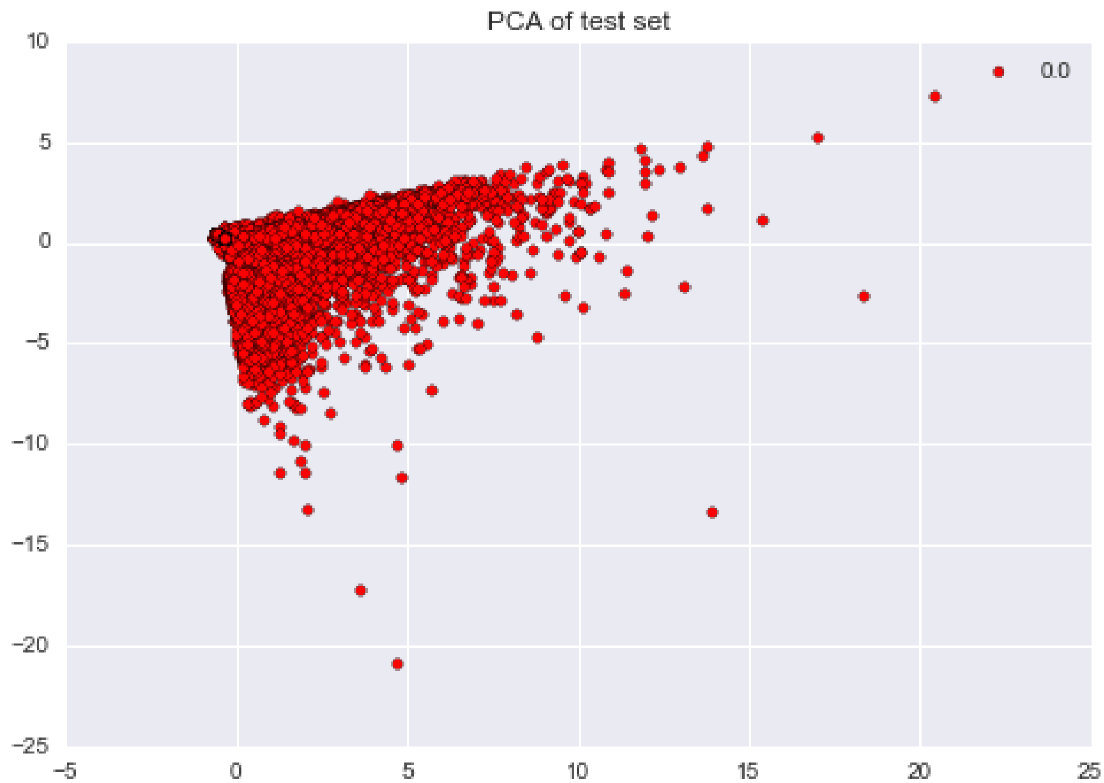
```
In [136]: X = train.drop(['TARGET'], axis=1)
          Xtest = test

          ## Standardize sets together
          Xtotal = X.append(Xtest)
          Xtotal_scaled = preprocessing.scale(Xtotal)
          X_scaled,Xtest_scaled = np.split(Xtotal_scaled, [ntrain])

          targets = np.array(train.TARGET)
          pca_plot(X_scaled, targets, title='PCA of training set')

          eggs = np.zeros((ntest,1))
          pca_plot(Xtest_scaled, eggs, title='PCA of test set')
```

```
explained variance ratio (first two components): [ 0.07887261  0.05153247]
explained variance ratio (first two components): [ 0.08508405  0.05360071]
```



0.4 Kernel PCA

This is very, very slow on the full dataset, and should not be run.

```
In [5]: #kernel_pca(X, targets, title='Kernel PCA of training set', kernel='poly', degree=3)
```

0.5 Create validation set

Since we have no test set to evaluate our performance on, we must create a validation set from a subset of our training set. This set will not be used for training and is NOT to be confused with the validation set used for cross-validation.

```
In [4]: from sklearn import cross_validation

        sss = cross_validation.StratifiedShuffleSplit(targets, 1, test_size=0.2, random_state=0)
        (train_index,valid_index) = list(sss)[0]

        X_train = X_scaled[train_index,]
        X_valid = X_scaled[valid_index,]
        y_train = targets[train_index]
        y_valid = targets[valid_index]

        print(X_train.shape)
        print(X_valid.shape)
```

```
(60816, 369)
(15204, 369)
```

0.6 Logistic regression

```
In [7]: from sklearn.linear_model import LogisticRegression

        def logreg_cv(X_train, y_train, c, fold):
            logreg = LogisticRegression(penalty='l2', dual=False, max_iter=100, C=c)
            cv_scores = cross_validation.cross_val_score(logreg, X_train, y_train, cv=fold)
            return np.mean(cv_scores)
```

```
In [ ]: ## Too slow

        #lambdas = np.power(float(2), range(-10,11))

        #result = [logreg_cv(X_train, y_train, c, 3) for c in lambdas]
```

As we can see, logistic regression with an L2-regularizer is way too slow on this dataset.

0.7 XGBoost

0.7.1 Logistic regression

```
In [ ]: import xgboost as xgb
        import re
        import StringIO

        def fpreproc(dtrain, dtest, param):
            label = dtrain.get_label()
            ratio = float(np.sum(label == 0)) / np.sum(label==1)
            param['scale_pos_weight'] = ratio
            return (dtrain, dtest, param)

        def parse_string(cv_string):
            a = cv_string.split("\t")[1]
            return re.search('(?<=:)[0-9.]+', a).group(0)



        lambdas = np.power(float(2), range(-10,11))
```

We use cross-validation to select the optimal value of lambda and the number of rounds. We can also select the optimal values of eta, max_depth, etc.

Since the evaluation function used in the Kaggle competition is the AUC, we use that here as well.

For time purposes, I've set the maximum num_round to 20. This can and should be increased for better results though.

```
In [69]: def xgb_cv_lambda(X_train, y_train, l, max_round=10):
             dtrain = xgb.DMatrix(X_train, y_train)

             param = {'bst:max_depth':2, 'bst:eta':1, 'silent':1, 'objective':'binary:logistic', 'lambda
             param['nthread'] = 4
             param['eval_metric'] = 'auc'
             num_round = max_round

             ## verbose_eval=False isn't available in my version of xgboost (0.4.0)
             actualstderr = sys.stderr
             sys.stderr = StringIO.StringIO()
```

```
        results = xgb.cv( param, dtrain, num_round, 3, fpreproc = fpreproc)
        sys.stderr = actualstderr
        sys.stderr.flush()

        test_errors = [float(i) for i in [parse_string(c) for c in results]]
        return [np.max(test_errors), np.argmax(test_errors)]


    cv_errors = [xgb_cv_lambda(X_train, y_train, l, 20) for l in lambdas]

    #bst.save_model('0001.model')

    #ypred = bst.predict(dtest,ntree_limit=bst.best_ntree_limit)
    #sum(y_valid == ypred)/float(len(y_pred))
```

NOTE: The best value of lambda selected is at the boundary ($2\hat{}10 = 1024$). When I have time, I should re-run cross-validation with a set of higher lambda values.

Having found our best parameter values, we train with those to get the desired model. Testing this out on our validation (**not test**) set, we get an error of:

```
In [131]: index = np.argmax([x[0] for x in cv_errors])
          bestlambda = lambdas[index]
          bestnrounds = cv_errors[index][1]+1

          def train_and_predict(X_train, y_train, X_test, y_test, l, existing_prediction = False, prepro
              dtrain = xgb.DMatrix(X_train, y_train)
              dtest = xgb.DMatrix(X_valid, y_valid)

              param = {'bst:max_depth':2, 'bst:eta':1, 'silent':1, 'objective':'binary:logistic', 'lamb
              param['nthread'] = 4
              param['eval_metric'] = 'auc'

              if preprocess == True:
                  dtrain, dtest, param = fpreproc(dtrain, dtest, param)

              evallist  = [(dtest,'eval'), (dtrain,'train')]

              ## Boost from existing prediction
              if existing_prediction == True:
                  actualstderr = sys.stderr
                  sys.stderr = StringIO.StringIO()
                  bst = xgb.train(param, dtrain, n, evallist)
                  sys.stderr = actualstderr
                  sys.stderr.flush()

                  tmp_train = bst.predict(dtrain, output_margin=True)
                  tmp_test = bst.predict(dtest, output_margin=True)
                  dtrain.set_base_margin(tmp_train)
                  dtest.set_base_margin(tmp_test)
                  bst = xgb.train(param, dtrain, n, evallist )
              else:
                  bst = xgb.train(param, dtrain, n, evallist)

In [135]: train_and_predict(X_train, y_train, X_valid, y_valid, bestlambda, existing_prediction=True, p
```

```
[0]	eval-auc:0.817792	train-auc:0.844025
[1]	eval-auc:0.817957	train-auc:0.844095
[2]	eval-auc:0.817870	train-auc:0.844364
[3]	eval-auc:0.818171	train-auc:0.844602
[4]	eval-auc:0.818321	train-auc:0.844689
[5]	eval-auc:0.818457	train-auc:0.844763
[6]	eval-auc:0.818606	train-auc:0.844976
[7]	eval-auc:0.818630	train-auc:0.845200
[8]	eval-auc:0.819045	train-auc:0.845416
[9]	eval-auc:0.819068	train-auc:0.845673
[10]	eval-auc:0.819671	train-auc:0.846085
[11]	eval-auc:0.819596	train-auc:0.846276
[12]	eval-auc:0.819485	train-auc:0.846303
[13]	eval-auc:0.819509	train-auc:0.846616
[14]	eval-auc:0.819741	train-auc:0.846761
[15]	eval-auc:0.819776	train-auc:0.846951
[16]	eval-auc:0.819878	train-auc:0.847038
[17]	eval-auc:0.819902	train-auc:0.847088
[18]	eval-auc:0.819830	train-auc:0.847207
[19]	eval-auc:0.820282	train-auc:0.847549
[20]	eval-auc:0.820378	train-auc:0.847637
[21]	eval-auc:0.820474	train-auc:0.847696
[22]	eval-auc:0.820405	train-auc:0.847987
[23]	eval-auc:0.820457	train-auc:0.848225
[24]	eval-auc:0.820458	train-auc:0.848326
[25]	eval-auc:0.820552	train-auc:0.848451
[26]	eval-auc:0.820589	train-auc:0.848456
[27]	eval-auc:0.820652	train-auc:0.848469
[28]	eval-auc:0.820720	train-auc:0.848523
[29]	eval-auc:0.820666	train-auc:0.848557
```

So we achieve a validation set AUC of 0.82 with minimal tuning.

```
In [ ]:
```