Create different feature sets

↳ *17 cells hidden*

Run the Classifiers

Load the data

```
def getDataAndLabels(name, features, label_encoder):
    labels_string = features.cancer_type

    labels        = label_encoder.fit_transform(labels_string)

    # Get rid of the cancer type and patient_barcode columns
    data = features[features.columns[3:]]

    return {'name': name, 'feature_size': data.shape[1],
            'data': data, 'labels': labels , 'label_encoder': label_encoder }
```

```
print('Loading training data ...')
# label encoder
label_encoder   = preprocessing.LabelEncoder()

# get all file names that start with features_
train_files = glob.glob("./data/features_*.train.csv")
all_train_data = {}

# load all of the files
for filename in train_files:

    name = filename[16:-10]
    print(" ", name)
    train_features = pd.read_csv(filename)
    all_train_data[name] = getDataAndLabels(name, train_features, label_encoder)

print("done.")
```

```
Loading training data ...
   topgenes_small
   bestfit_with_topgenes
   bestfit_med
   bestfit_large
   all
   l1reg_c0.5
   l1reg_c1
   l1reg_c10
   l1reg_c100
   after_pca
 done.
```

```
print('Loading test data ...')

test_files = glob.glob("./data/features_*.test.csv")
all_test_data = {}
for filename in test_files:

    name = filename[16:-9]
    #if (name != 'after_pca'):
    print(" ", name)
    test_features = pd.read_csv(filename)
    all_test_data[name] = getDataAndLabels(name, test_features, label_encoder)

print("done.")
```

```
Loading test data ...
   topgenes_small
   bestfit_with_topgenes
   bestfit_med
   bestfit_large
   all
   l1reg_c0.5
   l1reg_c1
   l1reg_c10
   l1reg_c100
   after_pca
 done.
```

Functions for running different classifiers

```
def getBestParamsLogit(train_data, train_labels):
    #
    # Logistic Regression
    #
    lr = LogisticRegression(penalty='l2', multi_class = 'ovr', solver='liblinear', max_iter=150)
    params = {'C': [0.1, 0.25,  0.5,]}
    logit = GridSearchCV(lr, params, cv=5,
                         scoring='accuracy', return_train_score=True)

    # Fit  training data
    logit.fit(train_data, train_labels)
    # Show the best C parameter to use and the expected accuracy
    print(' Best param:', logit.best_params_)
    print(' Accuracy:  ', np.round(logit.best_score_, 4) )

    return logit.best_params_
```

```
def getBestParamsSVM(train_data, train_labels):
    #
    # SVM
```

```
    #
    classifier = LinearSVC(penalty='l2')

    params = {'C': [0.01, 0.1, 0.5]}
    svm = GridSearchCV(classifier, params, cv=4,
                       scoring='accuracy', return_train_score=True)

    # Fit  training data
    svm.fit(train_data, train_labels)
    # Show the best C parameter to use and the expected accuracy
    print(' Best param:', svm.best_params_)
    print(' Accuracy:  ', np.round(svm.best_score_, 4) )

    return svm.best_params_


#
# Logistic regression
#
def run_logistic_regression(train_data, train_labels, test_data, test_labels, name, hyper_params):
    start = time.process_time()
    if name in hyper_params and 'lr' in hyper_params[name]:
        best_params_logit = hyper_params[name]['lr']
    else:
        print("Running grid search on Logistic Regression...")
        best_params_logit = getBestParamsLogit(train_data, train_labels)

    # Run logistic regression with L2 regularization on reduced
    # feature set
    lr = LogisticRegression(penalty='l2', tol=.01, max_iter=150,
                            C=best_params_logit['C'],
                            solver="liblinear", multi_class="ovr")
    lr.fit(train_data, train_labels)
    predict = lr.predict(test_data)
    elapsed_time = time.process_time() - start

    # Get precision, recall, f1 scores
    logit_prf_scores      = precision_recall_fscore_support(test_labels, predict, average='weighted')
    logit_scores_by_label = precision_recall_fscore_support(test_labels, predict, average=None)

    # Get confusion matrix
    logit_confusion       = confusion_matrix(test_labels, predict)

    print("\nLogistic Regression", name)
    print("  precision:", np.round(logit_prf_scores[0], 4))
    print("  recall:   ", np.round(logit_prf_scores[1], 4))
    print("  f1:       ", np.round(logit_prf_scores[2], 4))
    print("  time:     ", np.round(elapsed_time, 2))
    return {'scores': [
              logit_prf_scores[0],
              logit_prf_scores[1],
              logit_prf_scores[2],
              logit_scores_by_label,
              logit_confusion],
            'time': elapsed_time }


#
# Linear SVM
#
def run_linear_svm(train_data, train_labels, test_data, test_labels, name, hyper_params):
    print("\nLinear SVM", name)
    start = time.process_time()
    if name in hyper_params and 'svm' in hyper_params[name]:
        best_params_svm = hyper_params[name]['svm']
    else:
        print("Running grid search on Linear SVM...")
        best_params_svm = getBestParamsSVM(train_data, train_labels)

    svm = LinearSVC(penalty='l2', C=best_params_svm['C'])

    svm.fit(train_data, train_labels,)
    predict = svm.predict(test_data)
    elapsed_time = time.process_time() - start

    # Get precision, recall, f1 scores
    svm_prf_scores      = precision_recall_fscore_support(test_labels, predict, average='weighted')
    svm_scores_by_label = precision_recall_fscore_support(test_labels, predict, average=None)

    # Get confusion matrix
    svm_confusion       = confusion_matrix(test_labels, predict)

    print("  precision:", np.round(svm_prf_scores[0], 4))
    print("  recall:   ", np.round(svm_prf_scores[1], 4))
    print("  f1:       ", np.round(svm_prf_scores[2], 4))
    print("  time:     ", np.round(elapsed_time, 2))

    return {'scores': [
              svm_prf_scores[0],
              svm_prf_scores[1],
              svm_prf_scores[2],
              svm_scores_by_label,
              svm_confusion],
            'time': elapsed_time }


#
# Decision tree
#
def run_decision_tree(train_data, train_labels, test_data, test_labels, name, hyper_params):

    print("\nDecision Tree", name)
    start = time.process_time()
    dt = DecisionTreeClassifier()

    dt.fit(train_data, train_labels,)
    predict = dt.predict(test_data)
    elapsed_time = time.process_time() - start


    # Get precision, recall, f1 scores
    dt_prf_scores      = precision_recall_fscore_support(test_labels, predict, average='weighted')
    dt_scores_by_label = precision_recall_fscore_support(test_labels, predict, average=None)

    # Get confusion matrix
    dt_confusion       = confusion_matrix(test_labels, predict)


    print("  precision:", np.round(dt_prf_scores[0], 4))
    print("  recall:   ", np.round(dt_prf_scores[1], 4))
    print("  f1:       ", np.round(dt_prf_scores[2], 4))
```

```
    print("  time:     ", np.round(elapsed_time, 2))

    return {'scores': [
              dt_prf_scores[0],
              dt_prf_scores[1],
              dt_prf_scores[2],
              dt_scores_by_label,
              dt_confusion],
            'time': elapsed_time }


#
# Random forest
#
def run_random_forest(train_data, train_labels, test_data, test_labels, name, hyper_params):
    print("\nRandom Forest", name)
    start = time.process_time()
    rf = RandomForestClassifier(n_estimators=500)

    rf.fit(train_data, train_labels,)
    predict = rf.predict(test_data)
    elapsed_time = time.process_time() - start


    # Get precision, recall, f1 scores
    rf_prf_scores      = precision_recall_fscore_support(test_labels, predict, average='weighted')
    rf_scores_by_label = precision_recall_fscore_support(test_labels, predict, average=None)

    # Get confusion matrix
    rf_confusion       = confusion_matrix(test_labels, predict)

    print("  precision:", np.round(rf_prf_scores[0], 4))
    print("  recall:   ", np.round(rf_prf_scores[1], 4))
    print("  f1:       ", np.round(rf_prf_scores[2], 4))
    print("  time:     ", np.round(elapsed_time, 2))

    return {'scores': [
              rf_prf_scores[0],
              rf_prf_scores[1],
              rf_prf_scores[2],
              rf_scores_by_label,
              rf_confusion],
            'time': elapsed_time }


#
# Neural Net
#
def run_neural_net(train_data, train_labels, test_data, test_labels, name, hyper_params):
    print("\nNeural Net", name)
    tr_lab = to_categorical(train_labels)
    test_lab = to_categorical(test_labels)
    start = time.process_time()
    model = K.Sequential()
    model.add(Dense(2000, input_dim=train_data.shape[1], activation='relu',
                    kernel_regularizer=regularizers.l1_l2(l2=0.01,l1=0.01)))
    model.add(Dropout(0.2))
    model.add(Dense(1000, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(400, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(100, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(32, activation='sigmoid'))
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics = ["accuracy"])
    #model.fit(train_data, tr_lab, epochs=1000, batch_size=100)
    model.fit(train_data, tr_lab, epochs=100, batch_size=100)
    evaluate = model.evaluate(x = test_data, y = test_lab)
    predict = model.predict(test_data)
    elapsed_time = time.process_time() - start
    # Get precision, recall, f1 scores
    nn_prf_scores      = precision_recall_fscore_support(test_labels,np.argmax(predict,1), average='weighted')
    nn_scores_by_label = precision_recall_fscore_support(test_labels,np.argmax(predict,1), average=None)

    # Get confusion matrix
    #nn_confusion       = confusion_matrix(test_labels, predict)

    print("  precision:", np.round(nn_prf_scores[0], 4))
    print("  recall:   ", np.round(nn_prf_scores[1], 4))
    print("  f1:       ", np.round(nn_prf_scores[2], 4))
    print("  time:     ", np.round(elapsed_time, 2))

    return {'scores': [
              nn_prf_scores[0],
              nn_prf_scores[1],
              nn_prf_scores[2],
              nn_scores_by_label,
              []],  ## TODO Jeremy:  add confusion matrix
            'time': elapsed_time}


#
# XGBoost
#
def run_xg_boost(train_data, train_labels, test_data, test_labels, name, hyper_params):
    print("\nXG Boost", name)
    start = time.process_time()

    xgb_params = {
    'max_depth': 2,
    'eta': 0.3,
    'silent': False,
    'verbose': True,
    'objective': 'multi:softprob',
    'num_class': 32,
    'num_boost_round' : 2}

    xgb_cfr = xgb.XGBClassifier(**xgb_params)
    xgb_cfr.fit(train_data, train_labels)

    predict = xgb_cfr.predict(test_ata)
    elapsed_time = time.process_time() - start


    # Get precision, recall, f1 scores
    xgb_prf_scores      = precision_recall_fscore_support(test_labels, predict, average='weighted')
    xgb_scores_by_label = precision_recall_fscore_support(test_labels, predict, average=None)

    # Get confusion matrix
```

```
    xgb_confusion       = confusion_matrix(test_labels, predict)

    print("  precision:", np.round(xgb_prf_scores[0], 4))
    print("  recall:   ", np.round(xgb_prf_scores[1], 4))
    print("  f1:       ", np.round(xgb_prf_scores[2], 4))
    print("  time:     ", np.round(elapsed_time, 8))

    return {'scores': [
              xgb_prf_scores[0],
              xgb_prf_scores[1],
              xgb_prf_scores[2],
              xgb_scores_by_label,
              xgb_confusion],
            'time': elapsed_time}
```

Run the different classifiers

```
def get_saved_metrics():
    metrics_filename = "./data/metrics.csv"
    if os.path.isfile(metrics_filename):
        df_report_existing = pd.read_csv(metrics_filename)
        return df_report_existing
    else:
        return pd.DataFrame(columns=['name', 'classifier', 'feature_size', 'precision', 'recall', 'f1', 'time'])


def save_metrics(scores, times, save=True):
    df_scores = pd.DataFrame(scores)
    df_times = pd.DataFrame(times)
    rows = []

    for classifier in df_scores.index:
        for name in df_scores.columns:
            rows.append([
                        name,
                        all_train_data[name]['feature_size'],
                        classifier,
                        df_scores.loc[classifier][name][0],
                        df_scores.loc[classifier][name][1],
                        df_scores.loc[classifier][name][2],
                        df_times.loc[classifier][name]
                        ])

    df_report = pd.DataFrame(rows, columns=['name', 'feature_size', 'classifier', 'precision', 'recall', 'f1', 'time'])

    if save:
      # Write out scores as csv files
      print("\nWriting metrics ...")
      df_report.to_csv("./data/metrics.csv")
      print("done.")

    return df_report
```

```
hyper_params = {
    'l1reg_c0.5':           {'lr': {'C': 0.25}, 'svm': {'C': 0.01}},
    'l1reg_c1':             {'lr': {'C': 0.25}, 'svm': {'C': 0.01}},
    'l1reg_c10':            {'lr': {'C': 0.1},  'svm': {'C': 0.01}},
    'l1reg_c100':           {'lr': {'C': 0.25}, 'svm': {'C': 0.01}},
    'topgenes_small':       {'lr': {'C': 0.25}, 'svm': {'C': 0.01}},
    'bestfit_med':          {'lr': {'C': 0.1 }, 'svm': {'C': 0.01}},
    'bestfit_large':        {'lr': {'C': 0.1 }, 'svm': {'C': 0.01}},
    'all':                  {'lr': {'C': 0.25}, 'svm': {'C': 0.01}},
    'bestfit_with_topgenes':{'lr': {'C': 0.1 }, 'svm': {'C': 0.01}},
    'after_pca':            {'lr': {'C': 0.5 }, 'svm': {'C': 0.01}}
}

def is_existing_run(existing_runs, the_classifier, the_name):
    matches  = existing_runs[(existing_runs.classifier == the_classifier) & (existing_runs.name == the_name)]
    return matches.shape[0]

def copy_metrics(existing_runs, the_classifier, the_name):
    matches  = existing_runs[(existing_runs.classifier == the_classifier) & (existing_runs.name == the_name)]
    scores[the_name][the_classifier] = [matches.precision.values[0], matches.recall.values[0], matches.f1.values[0]]
    times[the_name][the_classifier]  = matches.time.values[0]

def run_classifiers(train_data, train_labels, test_data, test_labels, name, scores, times, existing_runs):
    scores[name] = {}
    times[name] = {}

    if is_existing_run(existing_runs, 'lr', name) == 0:
        lr = run_logistic_regression(train_data, train_labels, test_data, test_labels, name, hyper_params)
        scores[name]['lr']  = lr['scores']
        times[name]['lr']  = lr['time']
    else:
        copy_metrics(existing_runs, 'lr', name)


    if is_existing_run(existing_runs, 'svm', name) == 0:
        svm = run_linear_svm(train_data, train_labels, test_data, test_labels, name, hyper_params)
        scores[name]['svm'] = svm['scores']
        times[name]['svm'] = svm['time']
    else:
        copy_metrics(existing_runs, 'svm', name)


    if is_existing_run(existing_runs, 'dt', name) == 0:
        dt = run_decision_tree(train_data, train_labels, test_data, test_labels, name, hyper_params)
        scores[name]['dt']  = dt['scores']
        times[name]['dt']  = dt['time']
    else:
        copy_metrics(existing_runs, 'dt', name)


    if is_existing_run(existing_runs, 'rf', name) == 0:
        rf = run_random_forest(train_data, train_labels, test_data, test_labels, name, hyper_params)
        scores[name]['rf']  = rf['scores']
        times[name]['rf']  = rf['time']
    else:
        copy_metrics(existing_runs, 'rf', name)


    if is_existing_run(existing_runs, 'nn', name) == 0:
        nn = run_neural_net(train_data, train_labels, test_data, test_labels, name, hyper_params)
        scores[name]['nn']  = nn['scores']
        times[name]['nn']  = nn['time']
    else:
```

```
        copy_metrics(existing_runs, 'nn', name)


    save_metrics(scores, times)
```

```
scores = {}
times  = {}

existing_runs = get_saved_metrics()

for name in all_train_data.keys():
    print("************************")
    print(name)
    print("************************")

    train      = all_train_data[name]
    test       = all_test_data[name]

    run_classifiers(train['data'], train['labels'], test['data'], test['labels'], name, scores, times,
                    existing_runs)
```

Visualize Performance across different feature sets, different classifiers

```
df_report = save_metrics(scores, times, False)
df_report
```

	name	feature_size	classifier	precision	recall	f1	time
0	topgenes_small	8865	dt	0.355003	0.347652	0.345478	22.808223
1	bestfit_with_topgenes	5000	dt	0.343186	0.338661	0.336160	12.592472
2	bestfit_med	5000	dt	0.342441	0.340160	0.336631	12.373375
3	bestfit_large	8000	dt	0.357208	0.347652	0.347006	19.917533
4	all	19198	dt	0.344909	0.346653	0.339245	46.563513
5	l1reg_c0.5	4697	dt	0.336232	0.332168	0.330775	12.234848
6	l1reg_c1	7430	dt	0.341197	0.331169	0.331370	20.674227
7	l1reg_c10	13680	dt	0.346436	0.343157	0.338764	35.242824
8	l1reg_c100	18363	dt	0.358837	0.359141	0.352412	43.472572
9	after_pca	8006	dt	0.205518	0.191309	0.194230	155.755809
10	topgenes_small	8865	lr	0.549762	0.539960	0.519851	17.257251
11	bestfit_with_topgenes	5000	lr	0.539149	0.538961	0.515970	9.780974
12	bestfit_med	5000	lr	0.539149	0.538961	0.515970	9.743394
13	bestfit_large	8000	lr	0.554484	0.541459	0.519183	14.775475
14	all	19198	lr	0.549622	0.543956	0.521253	63.787119
15	l1reg_c0.5	4697	lr	0.536659	0.530470	0.510773	10.406144
16	l1reg_c1	7430	lr	0.544209	0.531968	0.512307	15.647111
17	l1reg_c10	13680	lr	0.565763	0.549451	0.523657	44.599937
18	l1reg_c100	18363	lr	0.550274	0.543457	0.521391	63.918611
19	after_pca	8006	lr	0.528760	0.501998	0.485617	863.198566
20	topgenes_small	8865	nn	0.481773	0.476523	0.458486	395.713549
21	bestfit_with_topgenes	5000	nn	0.497606	0.499001	0.479298	268.178483
22	bestfit_med	5000	nn	0.487686	0.486014	0.469875	229.497351
23	bestfit_large	8000	nn	0.510077	0.495504	0.480739	364.302666
24	all	19198	nn	0.512175	0.493007	0.483623	761.913399
25	l1reg_c0.5	4697	nn	0.468076	0.499500	0.470255	214.016822
26	l1reg_c1	7430	nn	0.000100	0.009990	0.000198	342.288193
27	l1reg_c10	13680	nn	0.000100	0.009990	0.000198	621.967364
28	l1reg_c100	18363	nn	0.519938	0.506494	0.492145	757.566261
29	after_pca	8006	nn	0.000100	0.009990	0.000198	418.716876
30	topgenes_small	8865	rf	0.449816	0.457043	0.410779	178.149539
31	bestfit_with_topgenes	5000	rf	0.454264	0.455544	0.414381	123.981630
32	bestfit_med	5000	rf	0.430347	0.458042	0.415872	124.758752
33	bestfit_large	8000	rf	0.440372	0.453546	0.409261	168.571105
34	all	19198	rf	0.446646	0.445554	0.403445	298.347052
35	l1reg_c0.5	4697	rf	0.476551	0.462038	0.422336	122.613206
36	l1reg_c1	7430	rf	0.474814	0.464036	0.422663	166.029324
37	l1reg_c10	13680	rf	0.462523	0.453047	0.411935	240.343188
38	l1reg_c100	18363	rf	0.441456	0.450549	0.404603	290.577816
39	after_pca	8006	rf	0.137010	0.140360	0.091183	742.582230
40	topgenes_small	8865	svm	0.531526	0.532967	0.510535	3.671322
41	bestfit_with_topgenes	5000	svm	0.508267	0.518981	0.497027	2.395096
42	bestfit_med	5000	svm	0.508267	0.518981	0.497027	2.326138
43	bestfit_large	8000	svm	0.537923	0.535964	0.517101	3.330731
44	all	19198	svm	0.538346	0.544456	0.520709	6.428361
45	l1reg_c0.5	4697	svm	0.526820	0.535465	0.512240	2.253765
46	l1reg_c1	7430	svm	0.526239	0.535465	0.513322	3.134554
47	l1reg_c10	13680	svm	0.525074	0.534965	0.509875	5.420964
48	l1reg_c100	18363	svm	0.538732	0.544456	0.520623	6.368480
49	after_pca	8006	svm	0.516078	0.515485	0.490608	289.964595

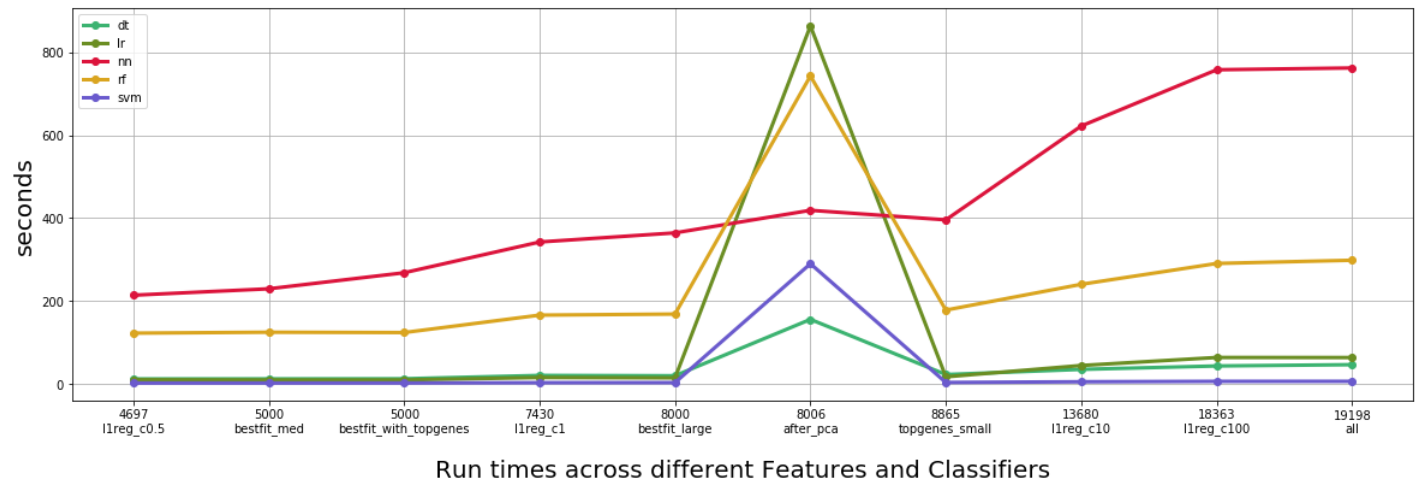
```
colors = {'lr': 'olivedrab', 'svm': 'slateblue',
          'dt': 'mediumseagreen', 'rf': 'goldenrod',
          'xgb': 'coral', 'nn': 'crimson'}

def plot_classifier_metrics(df_report, label_encoder):

    plt.rcParams["figure.figsize"] = (20,20)
```

```
    labels = []
    for key, group in df_report.groupby(['feature_size', 'name']):
        labels.append(str(key[0]) + '\n' + key[1])

    sorted_df_report = df_report.sort_values(by=['classifier', 'feature_size', 'name'], ascending=[1,1,1])



    for classifier, group in sorted_df_report.groupby(['classifier']):

        plt.plot(labels, group.precision.values, color=colors[classifier],
                 linewidth=3, label=classifier + " precision", marker='o' )
        plt.plot(labels, group.recall.values, color=colors[classifier], linestyle="dashed",
                 linewidth=3, label=classifier + " recall", marker='o' )


    plt.yticks(np.arange(0, .65, .01))
    plt.ylabel('Precision, Recall', fontsize=20)
    plt.xlabel('Precision and Recall across different Features and Classifiers', fontsize=20, labelpad=20)
    plt.legend()
    plt.grid()
    plt.show()


def plot_classifier_times(df_report, label_encoder):

    plt.rcParams["figure.figsize"] = (20,6)

    labels = []
    for key, group in df_report.groupby(['feature_size', 'name']):
        labels.append(str(key[0]) + '\n' + key[1])

    sorted_df_report = df_report.sort_values(by=['classifier', 'feature_size', 'name'], ascending=[1,1,1])



    for classifier, group in sorted_df_report.groupby(['classifier']):

        plt.plot(labels, group.time.values, color=colors[classifier],
                 linewidth=3, label=classifier, marker='o' )



    plt.ylabel('seconds', fontsize=20)
    plt.xlabel('Run times across different Features and Classifiers', fontsize=20, labelpad=20)
    plt.legend()
    plt.grid()
    plt.show()


def show_precision_recall_by_label(precision_by_label, recall_by_label, name, classifier, label_encoder):
    plt.rcParams["figure.figsize"] = (20,20)
    labels = []
    for i in range(len(precision_by_label)):
        label = label_encoder.inverse_transform([i])[0]
        labels.append(label)

    y_pos = np.arange(len(labels))

    fig, (ax1, ax2) = plt.subplots(ncols=2, sharey=False)

    ax1.invert_xaxis()
    ax1.yaxis.tick_right()

    ax1.set_yticks(y_pos)
    ax1.set_yticklabels(labels)

    ax2.set_yticks(y_pos)
    ax2.set_yticklabels(labels)

    ax1.barh(y_pos, precision_by_label, color=colors[classifier] , label="precision")
    ax2.barh(y_pos, recall_by_label,    color=colors[classifier],  label='recall')

    ax1.set_title('Precision( ' + classifier + ')')
    ax2.set_title('Recall (' + classifier + ')')

    plt.grid()
    plt.show()


def coords_of_max(theArray, n):
    # Flatten the 2D array
    flat = theArray.flatten()
    # Partition so that the we know the sort order for
    # the cells with the highest values.  We just
    # care about the top n highest values.  So for example,
    # if n = 3, get return 3 indices.
    indices = np.argpartition(flat, -n)[-n:]
    # Reverse so that we show index of highest value first
    # (descending)
    indices = indices[np.argsort(-flat[indices])]
    # Now return the coordinates for these indices
    # for a 2D array.  This will return 2 arrays,
    # the first for the row index, the second for the
    # column index.  The row index represents the
    # actual digit, the column index represents
    # the confused digit
    return np.unravel_index(indices, theArray.shape)


def show_confusion_matrix(conf_mx, label_encoder):
    # Determine the error rates for each misclassification pair
    row_sums = conf_mx.sum(axis=1, keepdims=True)
    norm_conf_mx = conf_mx / row_sums
    # Set the error rates for correctly classified pairs (the diagonal) to zero
    np.fill_diagonal(norm_conf_mx, 0)

    max_coords = coords_of_max(norm_conf_mx, 20)
    confusion_rows = []
    for i in range(len(max_coords[0])):

        # This is the actual label
        actual_label_idx  = max_coords[0][i]
        actual_label      = label_encoder.inverse_transform([actual_label_idx])[0]

        # This is the predicted label
        predicted_label_idx = max_coords[1][i]
        predicted_label = label_encoder.inverse_transform([predicted_label_idx])[0]

        # This is the error rate
```

```
        error_rate  = norm_conf_mx[max_coords[0][i], max_coords[1][i]]
        error_count = conf_mx[max_coords[0][i], max_coords[1][i]]

        row = list([ actual_label,
                     predicted_label,
                     code_to_disease[actual_label][0],
                     code_to_disease[predicted_label][0],
                     error_rate,
                     error_count ])
        confusion_rows.append(row)

    df = pd.DataFrame(confusion_rows, columns=['actual', 'predicted',  'actual_name', 'predicted_name', 'error_rate', 'error_count'])
    display(df)
```

Plot precision metrics across different classifiers and feature sets

```
# Plot precision and accuracy across different classifiers
plot_classifier_metrics(df_report, label_encoder)

# Plot time across different classifiers
plot_classifier_times(df_report, label_encoder)
```



Report the precision, recall, and f1 score across different classifiers and feature sets

```
# best precision
```

```
sorted_df = df_report.sort_values(by='precision', ascending=0)
best_precision = sorted_df.head(1)

# best recall
sorted_df = df_report.sort_values(by='recall', ascending=0)
best_recall = sorted_df.head(1)

# best f1
sorted_df = df_report.sort_values(by='f1', ascending=0)
best_f1 = sorted_df.head(1)

# Show the feature set and classifier with the best
# precision, recall, and f1 scores
print("\n\nBest precision")
display(best_precision)
print("\n\nBest recall")
display(best_recall)
print("\n\nBest f1")
display(best_f1)

# get the scores by label and confusion matrix
# for the best prediction
best_prediction = best_precision
best_name       = best_prediction.name.values[0]
best_classifier = best_prediction.classifier.values[0]
precision_by_label = scores[best_name][best_classifier][3][0]
recall_by_label = scores[best_name][best_classifier][3][1]
best_confusion_matrix = scores[best_name][best_classifier][4]

# show a side-by-side barchart of precision and recall for each label
print("\n\nPrecision and Recall by Label for classifier ")
print("Classifier:", best_classifier, "Feature set:", best_name)
show_precision_recall_by_label(precision_by_label, recall_by_label,
                               best_name, best_classifier, label_encoder)
```

Best precision

	name	feature_size	classifier	precision	recall	f1	time
17	l1reg_c10	13680	lr	0.565763	0.549451	0.523657	44.599937

Best recall

	name	feature_size	classifier	precision	recall	f1	time
17	l1reg_c10	13680	lr	0.565763	0.549451	0.523657	44.599937

Best f1

	name	feature_size	classifier	precision	recall	f1	time
17	l1reg_c10	13680	lr	0.565763	0.549451	0.523657	44.599937

Precision and Recall by Label for classifier
Classifier: lr Feature set: l1reg_c10

Show the confusion matrix for the best performing classifier/feature set

```
show_confusion_matrix(best_confusion_matrix, label_encoder)
```

	actual	predicted	actual_name	predicted_name	error_rate	error_count
0	READ	COAD	Rectum_adenocarcinoma	Colon_adenocarcinoma	0.720000	18
1	ACC	BRCA	Adrenocortical_carcinoma	Breast_invasive_carcinoma	0.500000	10
2	PCPG	PRAD	Pheochromocytoma_and_Paraganglioma	Prostate_adenocarcinoma	0.413793	12
3	TGCT	PRAD	Testicular_Germ_Cell_Tumors	Prostate_adenocarcinoma	0.380952	8
4	KICH	BRCA	Kidney_Chromophobe	Breast_invasive_carcinoma	0.352941	6
5	MESO	PRAD	Mesothelioma	Prostate_adenocarcinoma	0.333333	7
6	PCPG	THCA	Pheochromocytoma_and_Paraganglioma	Thyroid_carcinoma	0.310345	9
7	UCS	UCEC	Uterine_Carcinosarcoma	Uterine_Corpus_Endometrial_Carcinoma	0.307692	4
8	DLBC	STAD	Lymphoid_Neoplasm_Diffuse_Large_B-cell_Lymphoma	Stomach_adenocarcinoma	0.300000	3
9	CESC	BRCA	Cervical_squamous_cell_carcinoma_and_endocervi...	Breast_invasive_carcinoma	0.290909	16
10	THYM	BRCA	Thymoma	Breast_invasive_carcinoma	0.280000	7
11	SARC	BRCA	Sarcoma	Breast_invasive_carcinoma	0.269231	14
12	KIRP	BRCA	Kidney_renal_papillary_cell_carcinoma	Breast_invasive_carcinoma	0.262295	16
13	MESO	KIRC	Mesothelioma	Kidney_renal_clear_cell_carcinoma	0.238095	5
14	MESO	BRCA	Mesothelioma	Breast_invasive_carcinoma	0.238095	5
15	TGCT	THCA	Testicular_Germ_Cell_Tumors	Thyroid_carcinoma	0.238095	5
16	UCS	BRCA	Uterine_Carcinosarcoma	Breast_invasive_carcinoma	0.230769	3
17	UCS	OV	Uterine_Carcinosarcoma	Ovarian_serous_cystadenocarcinoma	0.230769	3
18	OV	BRCA	Ovarian_serous_cystadenocarcinoma	Breast_invasive_carcinoma	0.223684	17
19	SARC	PRAD	Sarcoma	Prostate_adenocarcinoma	0.211538	11

Write out the confusion matrix and precision/recall by label to file

```
df_confusion_matrix = pd.DataFrame(best_confusion_matrix)
df_precision_by_label = pd.DataFrame(precision_by_label)
df_recall_by_label = pd.DataFrame(recall_by_label)

print("\nWriting metrics ...")
df_confusion_matrix.to_csv("./data/metrics_confusion_matrix.csv")
print("done.")

print("\nWriting metrics ...")
df_precision_by_label.to_csv("./data/metrics_precision_by_label.csv")
print("done.")

print("\nWriting metrics ...")
df_recall_by_label.to_csv("./data/metrics_recall_by_label.csv")
print("done.")
```

```
Writing metrics ...
done.

Writing metrics ...
done.

Writing metrics ...
done.
```