Project title: Breast cancer diagnosis prediction & experimental exploration
Project Type: System design based on real-world data and some experimental exploration.
Number of Student authors: 1
Chengyao Wang
chengyao@usc.edu
Dec. 1st, 2019

1 Abstract:

The main part of this project dedicates in conducting binary classification task on the famous 'Breast Cancer Wisconsin Dataset' from UCI repo, to predict the diagnosis of a single patient. CART based Boosting/Boosting Forest & Elastic Net Logistic regression & K means + Logistic regression are tuned for best achieving best accuracy. The best performed model is Elastic Net logistic regression with a 98%/97.48% accuracy in CV/test. The second part of the project is focused on exploring contribution of labels in a supervised learning problem by implementing semi-supervised/unsupervised methods for the original problem.

2 Introduction:

2.1 Problem Type, Statement, and Goals:

The 'Breast Cancer Wisconsin Dataset' is built aimed to predict the diagnosis of breast cancer of individuals. Features are extracted from digitized image of a fine needle aspirate (FNA) of a breast mass. They describe the characteristics of the cell nuclei present in the image. Under the scope of this project, the problem on top of this dataset can be abstracted into a supervised binary classification problem, which tries to predict the diagnosis 'Malignant/Benign' of individuals. Furthermore, the binary labels also provide a great chance to explore the performance of semi-supervised/unsupervised method in classification scenarios, which is the second part of this project. This dataset has 569 labeled instances, with a 357/212 class ratio, and 30 features per-instance.

This dataset is forged in the 1990s, aiming to improve & assist in the diagnosis of breast cancer, which is inherently important by its topic & goal. This dataset suffers neither from sparsity nor missing data. Irrelevance and correlatedness of features remain a problem, as shown in the data preprocessing part later. Also, from today's point of view, this dataset also suffers from limited number of training samples.

2.2 Literature Review and previous work:

Previous works are limited in numbers, mostly conducts classification on subsets of features. Cited by dataset's description, two well-known approach are Multi-surface Method-Tree [1] & LDA [2]. The former is a classification method which uses linear programming to construct a decision tree. Relevant features were selected using an exhaustive search in the space of 1-4 features and 1-3 separating planes. The latter is an actual linear program used to obtain the separating plane in the 3-dimensional space. Best 10-fold cross validation result on previous work are reported to be 97.5%.

2.3 Our Prior and Related Work:

Prior and Related Work – None.

2.4 Overview of Our Approach:

Preprocessing & visualization techniques:

Scatter/Box plots features with the top-5 CV (Coefficient of Variation), standardization, feature correlation heat matrix, per-feature class distribution plotting, linear/kernel PCA.

Part I, finding best performance models:

K-nearest neighbor, AdaBoost & Boosting Forest, Elastic net logistic regression, K means clustering + logistic regression.

Part II, experimental explorations:

Monte Carlo semi-supervised l1 support vector machine, Spectral Clustering.

Other techniques:

5-fold cross-validation with accuracy as criterion for model selection.
Hierarchical grid search for tuning tree-based models.

3. Implementation:

3.1 Data set:

The original dataset consists of 569 instances and 32 features. The first two features are ID & labels of each individual, thus is separated from the dataset from the beginning. As afore mentioned, the remaining 30 features in the dataset are all continuous, real-valued features of the cell nucleus. They are listed in the chart below, as well as a short description:

Feature Name	Description	Feature Name	Description
radius	mean of distances from center to points on the perimeter	compactness	perimeter^2 / area - 1.0
texture	standard deviation of gray-scale values	concavity	severity of concave portions of the contour
perimeter		concave	number of concave portions of the contour
area		symmetry	
smoothness	local variation in radius lengths	fractal dimension	("coastline approximation" - 1

One image of an individual contains several such observable cell nucleus, thus 30 features that are presented in the dataset come from statistical feature extraction, which are mean / std / largest respectively from the 10 features above, e.g. feature 3 is radius mean, 13 being radius STD, and 23 as largest radius. The precision of floating points is limited to four significant digits.

3.2 Preprocessing, Feature Extraction, Dimensionality Adjustment:

The generalized 'preprocessing' stage consists of the following steps, some will be discussed in detail in the subsections:

1. Read dataset, discard useless information "ID", split features/labels.
2. Use pseudo random number generator for train/test split with ratio roughly 4:1.
3. Perform standardization on the training set and spread the coefficients to test set.
4. Plot correlation heat matrix on both raw/standardized dataset.
5. Plot Scatter/Box plots features with the top-5 CV (Coefficient of Variation) with raw dataset.
6. Per-feature class distribution plotting, for coarse feature selection.
7. Perform linear/kernel PCA, for further dimension reduction.

3.2.1 Standardization & Correlation Matrix visualization / analysis[1]:

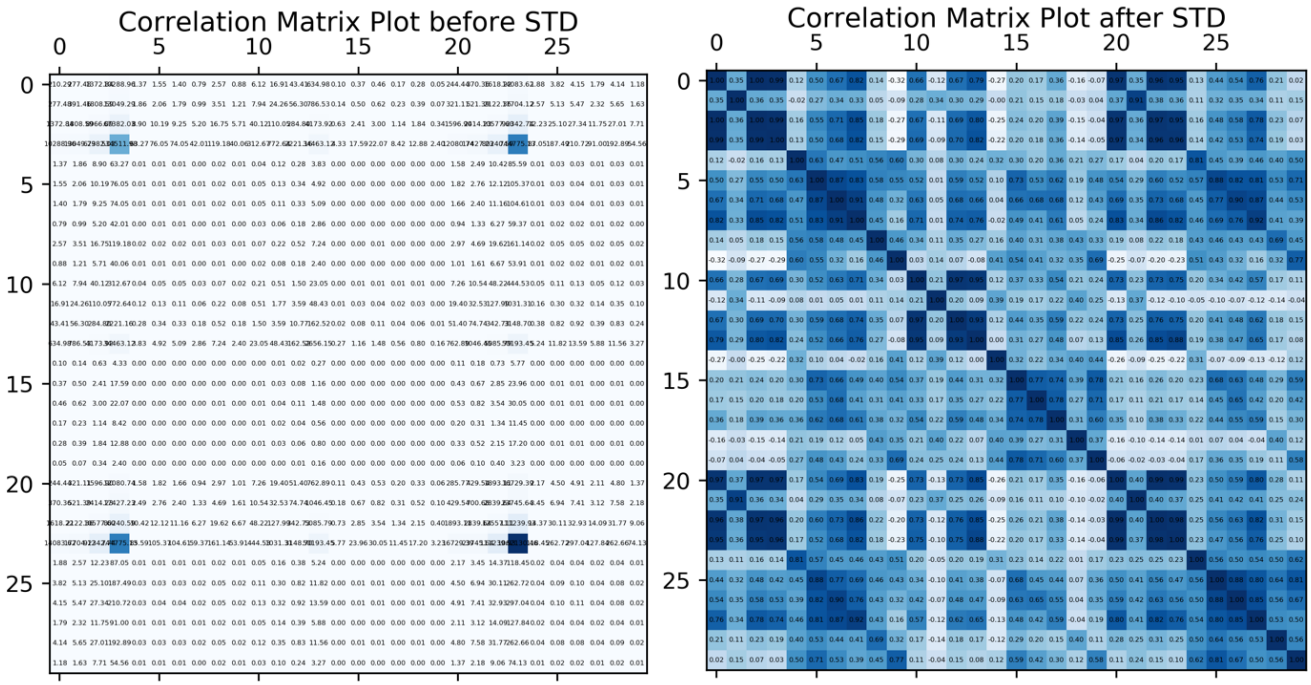
Standardization is a non-universal however common practice in data preprocessing. One good thing it does for feature correlation plotting is silencing the impact from different magnitudes of feature values, and scaling the correlation coefficient to [-1,1] as guaranteed by Cauchy-Schwartz inequality:

$$-\mathbf{1} \leq \frac{< \boldsymbol{X}, \boldsymbol{Y} >}{\|\boldsymbol{X}\|\|\boldsymbol{Y}\|} \leq \mathbf{1}$$

Sample mean/standard deviation are first calculated per-feature wise on the training set. Then, both train & test set are standardized according to the two coefficients. While generally we are not allowed to peek the test set before testing, the statistical feature calculated above doesn't include any information about the test set, thus is not considered as data snooping. Such normalizing the test set at the beginning are done for the sake of convenience for test phase.

Correlation matrix reveals pairwise correlation between different features from a linear dependency perspective. It, combined with heat map result, is shown below. Also, just for illustration, correlation matrix before standardization is also shown.

[1] Library used: matplotlib.pyplot, generated by dataGen.correlationMx()



The left figure clearly justifies the necessity for standardization before plotting the correlation matrix. And the right one proves the existence of the highly correlated features, but the numbers are limited. One can find several 4*4 color patterns in the plot, located in 0~4, 10~14, 20~24. Recalling that they are all extracted statistically from the first four of ten primal features, we can draw conclusions that 'radius/texture/perimeter/area' are highly correlated.

3.2.2 Coefficient of Variation analysis[2]:

Coefficient of variation (CV), aka relative standard deviation (RSD), is a standardized measure of dispersion of a probability distribution or frequency distribution. It is defined by:

$$c_v = \frac{\sigma}{\mu}$$

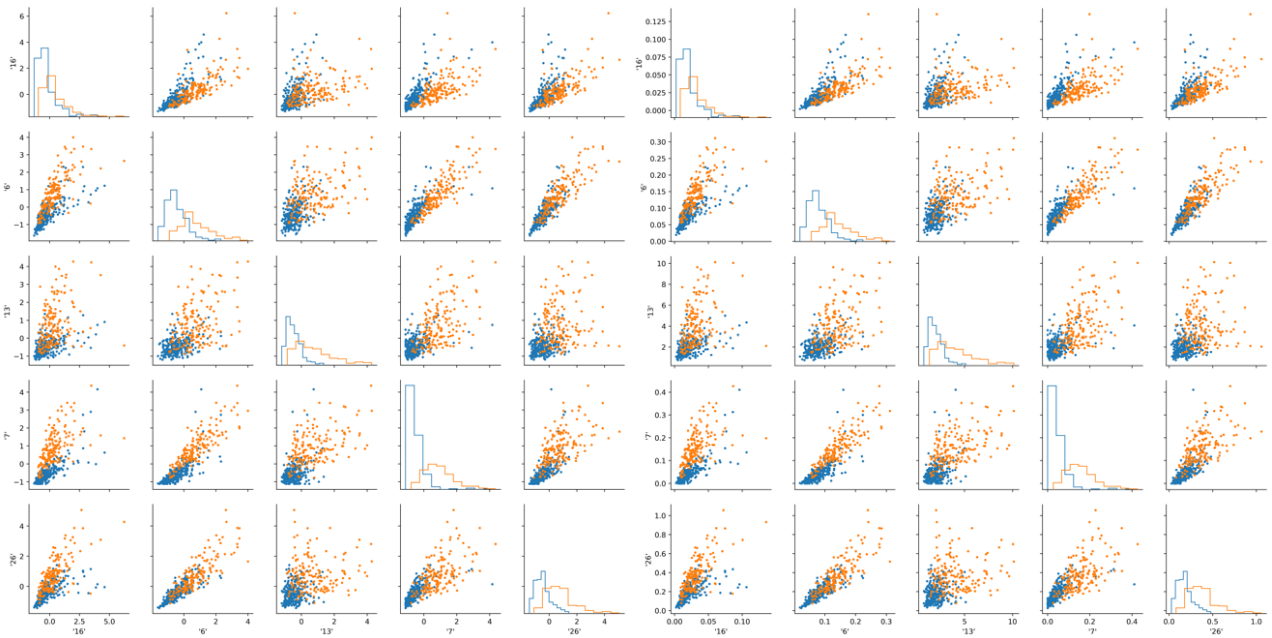
It shows the extent of variability in relation to the mean of the population. It exists as an alternative to standard deviation, and gains advantage especially when comparing features with different units or widely different means [3]. CV analysis always come before dataset standardization, or else the ratio will explode to infinity. It provides a good way to choose features to visualize when it's impossible to visualize all features, whose underlying philosophy being similar to PCA – more variance potentially preserves more information. But it doesn't provide any guidance for determining the significance of features and feature selection. It's a common practice to visualize $top - \sqrt{\#_{total\ features}}$ features, in this case top 5. The CV values for all 30 features are:

Feature	CV-value	Feature	CV-value	Feature	CV-value	Feature	CV-value	Feature	CV-value
0	0.24627	6	0.90421	12	0.68562	18	0.38837	24	0.17473
1	0.21988	7	0.79099	13	1.04862	19	0.71279	25	0.64036
2	0.26062	8	0.14807	14	0.43798	20	0.29338	26	0.78531
3	0.52654	9	0.11496	15	0.71941	21	0.23509	27	0.57443
4	0.14536	10	0.66120	16	0.98959	22	0.30901	28	0.21620
5	0.51178	11	0.43476	17	0.52415	23	0.62837	29	0.22657

Features selected for Coefficient Variation Plotting: ["'13'", "'16'", "'6'", "'7'", "'26'"]

The scatter/box plots are shown below, using raw/standardized dataset respectively:

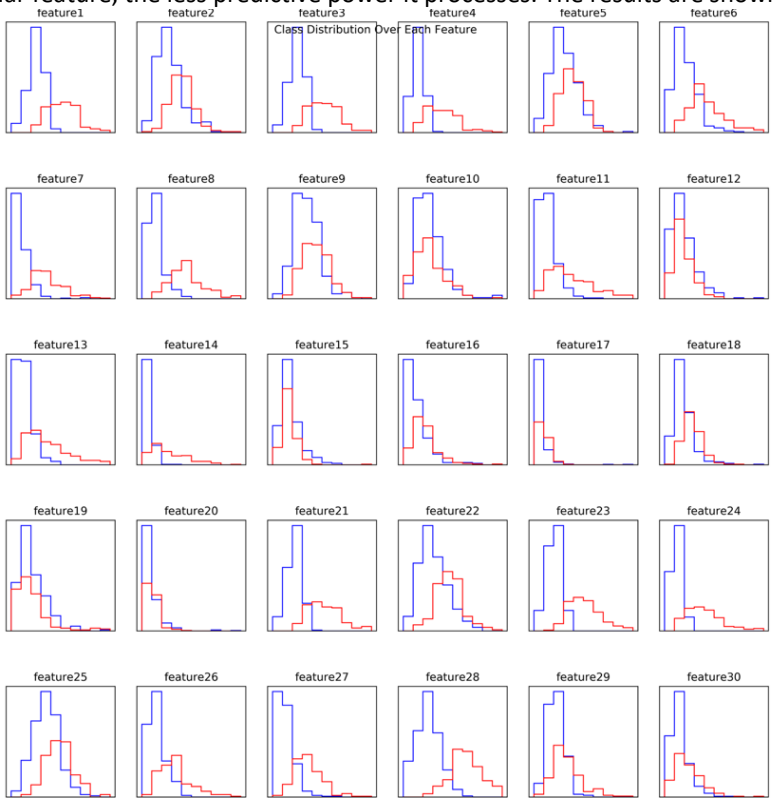
[2] Library used: seaborn, generated by util.scatterBoxPlot(), dataGen.CV_feature_plot()



It can be shown that, except feature '16', two classes demonstrated quite well separateness in each of the rest 4 features, either from box plots or pair scatter plots. Thus, we are facing a rather well-behaved dataset and expecting to get good classification performance.

3.2.3 Coarse feature selection by per-feature class distribution plotting[3]:

Coarse feature selection in this project is conducted under the same philosophy as Fisher-Linear Discriminant Analysis (LDA), which tries to evaluate the significance of features by how separated the means of different distributions of two classes are [4]. Moreover, one can also judge by the similarities of the class distributions, like Chi-square / K-S goodness-of-fit (GOF) test. Of course, the closer the means/the smaller GOF test results are, the more similar distributions classes have on one particular feature, the less predictive power it processes. The results are shown below:



[3] Library used: matplotlib.pyplot, generated by dataGen.feature_selection(), util.coarseSelectionPlot(), util.pcaPlot()

We can tell by observation that the two classes have very similar distributions over feature '10', '12', '15', '19', '20', in both mean and distribution. Thus, they are discarded and will not participate in all the steps following. Note that such 'empirical' judgement should be rather conservative and leave those equivocal features to PCA.

3.2.4. Linear & kernel PCA[4]:

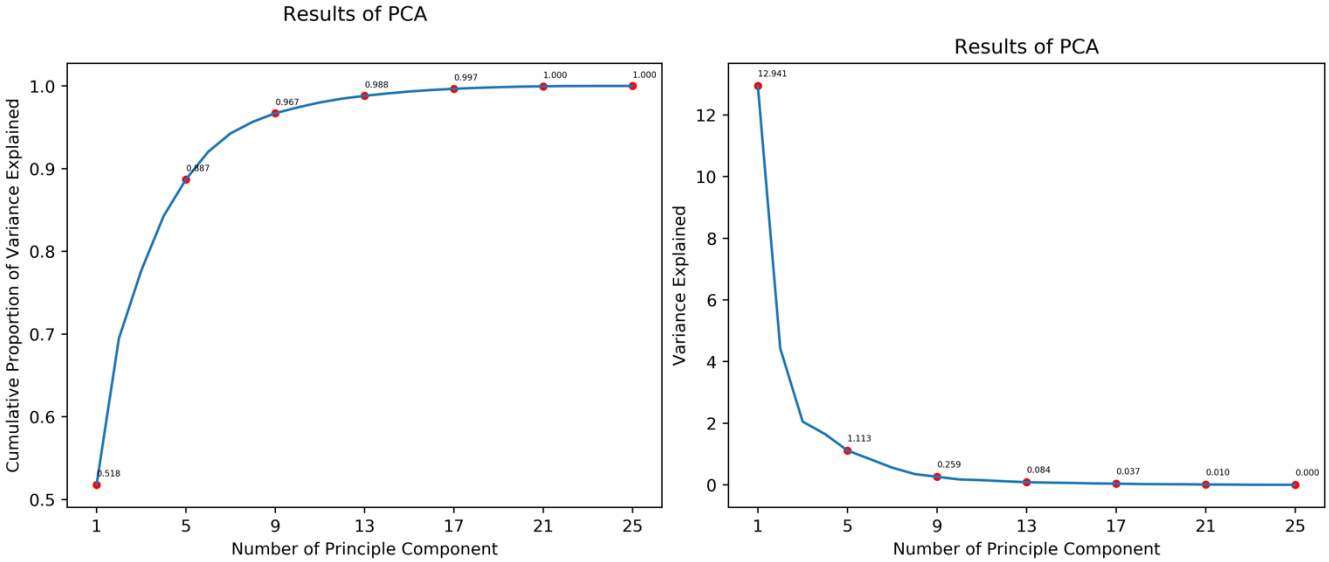
Principle Component Analysis (PCA) is a dimension reduction method based on empirical variance decomposition. As afore mentioned, PCA interprets variance as information and tries to project features to a lower dimensional space while preserving its variance to the most. Such process is done by Eigen-decomposition/SVD on the dataset matrix X, and naturally leads to the argument that new features of the projection space is a linear combination of the original features.

The target matrix L PCA is working on can be expressed as:

$$\boldsymbol{L = X^T X, L_{ij} = f_i^T f_j = \langle f_i, f_j \rangle, where\ f_i\ is\ the\ column\ vector\ of\ feature\ i}$$

The expression of Euclidean inner product in linear PCA make it easily extendable to 'kernel methods', which introduces kernel PCA. Like "kernel machines", it may help downstream models in getting a better performance as preprocessing method. But aside from the fact that it also may not help, it's not that trivial for kernel PCA to explicitly show 'how much variance explained by principle axis' as linear PCA does, causing problems to deciding to what extent should dimension reduction be done. Kernel chosen by kernel PCA is "RBF" kernel, and we don't take it as hyperparameter that needs cross validated on.

PCA results on top of the reduction by 3.2.3 is shown below:



The first 17 principle axis preserves 99.7% of all the variance, thus is selected as the dimension of the projected feature space. Like said, because we have no access to such curves for kernel PCA, the choice of projected space dimension is also set 17.

3.3 Dataset Methodology:

The train/test split process is done in a Monte-Carlo manner, and is self-implemented. Every data point has a 0.8 probability to be in train set, thus the final size ratio of two datasets may not be strictly 4:1. Splitting process finalized in 450 training set size & 119 test set size. Class ratios in both train/test set are reported as a reference, because some algorithms like logistic regression are known to be sensitive to class imbalance, but this information is not considered anywhere in this project before the test phase.

Due to the lack of dataset size, all cross validation is conducted in 5-fold and there is no validation set and pre-training set. Validation error are substituted by cross-validation error in all models covered in this project, and cross-validation error/accuracy serves as the criterion for model selection. We are not suffering extreme class imbalance from the dataset, thus we only consider accuracy during model selection. However, accuracy/precision/recall/confusion matrix are all reported on the test dataset. A good reason for using cross validation error instead of validation error came about during baseline model fitting, where the baseline model is greatly impacted by train/valid split. Averaging across folds helps

[4] Library used: sklearn.decomposition.PCA/KernelPCA, generated by dataGen.feature_selection()

lowering the variance. Single loop cross validation is used when tuning hyperparameters for the model once per-hyperparameter per-value, or pair if it's a grid search[5]. Detailed cross validation procedure for different models are introduced in the following section.

The test set are only used in the testing phase that come at the very end of this project for out-of-sample error estimation once per model and there are no decisions made based on the test results.

3.4 Training Process & Model Selection:

3.4.1 Baseline model: K nearest neighbor[6]:

We choose KNN for the baseline model with k = 5, because of its simplicity. 'Euclidean distance' are adopted as metric measure. The two hyperparameters are chosen by heuristics, and actually doesn't matter much since it's just a baseline model to get a subjective sense of the difficulty of the problem. Worth mentioning, KNN exploited a huge variance between different train-valid splits, where the validation accuracy ranged from some 92% to 98%. Thus, mean accuracy over 5-fold cross validation are used as an estimate of validation accuracy, which is expected to be more stable because of the averaging. The results for KNN are shown below:

```
Starting Baseline Model Fit using KNN = 5
Model is Vulerable to Train/Valid Splitting, Using CV = 5 to estimate validation Acc.
With Dataset After Vanilla PCA:
[0.978021978021978, 0.967032967032967, 0.989010989010989, 0.9325842696629213, 0.9775280898876404]
Fit Complete: Accuracy Mean = 0.96884, Accuracy STD = 0.02170
With Dataset After Kernel PCA:
[0.978021978021978, 0.978021978021978, 0.9230769230769231, 0.9775280898876404, 0.9662921348314607]
Fit Complete: Accuracy Mean = 0.96459, Accuracy STD = 0.02374
```

Baseline KNN achieved quite a high absolute training accuracy, thus we should be expecting later models to achieve better performance. Additionally, from the results, it's observed that referring to kernel PCA doesn't seem to help in a better performance, thus later models are all built & chosen on dataset preprocessed by linear PCA.

3.4.2 CART based model, AdaBoost & Boosting Forest[7]:

(a) Model & Method Description:

Ensemble methods are well known for their excellence in boosting the performance of base classifiers. They work especially well with CART, because of the simplicity and flexibility of CART. By limiting the capacity/complexity of individual CARTs to be 'decision stumps', boosting methods will be able to achieve low bias/variance, making them hard to overfit. AdaBoost is implemented & tuned in cooperation with bagging over multiple boosting trees, where 'Boosting-Forest' gets its name. The main idea is to see if aggregated ensemble methods are able to achieve a better performance. As a side product, individual boosting tree from the forest will be isolated and be evaluated. But its' worth mentioning that boosting trees are not tuned independently of the forest, thus can't be expected to reach best performance when tested alone.

As one of the setbacks resulted from the flexibility of tree-based methods, the excessive amount of hyperparameters often cause trouble in model selection, because they often came interweaved and time complexity of grid search grow exponentially with the dimension of the hyperparameter space. Thus, we adopted a hierarchical grid search approach, in compromise of performance and time. The overall order is determined by the importance of the hyperparameters and the role they play in the model. They are tuned mostly in pair. We slightly modified the original approach introduced by [5], to fit into our Boosting-Forest Schema and the dataset we're facing, e.g. size and number of features. Below is a high-level flow chart, and they will be further discussed in the next section.

$$\boldsymbol{Coarse\ Tune\ Boosting} \rightarrow \boldsymbol{Fine\ Tune\ CART} \rightarrow \boldsymbol{Fine\ Tune\ Boost} \rightarrow \boldsymbol{Fine\ Tune\ Bagging}$$

(b) Training Process & Model Selection:

The set of hyperparameters, or topological structure of the Boosting-Forest to tune are first determined if they are of our interest and are worth putting into cross validation. As an example, the boosting process can be interpreted as gradient descent in the functional space, thus is compatible with 'early stopping techniques' extensively used in neural networks mainly to combat overfitting. But here in this case, boosting is known for its robustness to overfitting by the nature of weak learners it's aggregating, and the model capacities of boosting trees are substantially lower than neural

[5] All cross validation routines are self-implemented, and they are embedded in models.XXX_XXX_main()

[6] Library used: sklearn.neighbors.KNeighborClassifier, generated by models.baselineFit()

[7] Library used: sklearn.ensemble.GradientBoostingClassifier, BaggingClassifier, generated by models.AccSqueeze_tree_main(), AccSqueeze_tree_last(), AccSqueeze_tree_toy()

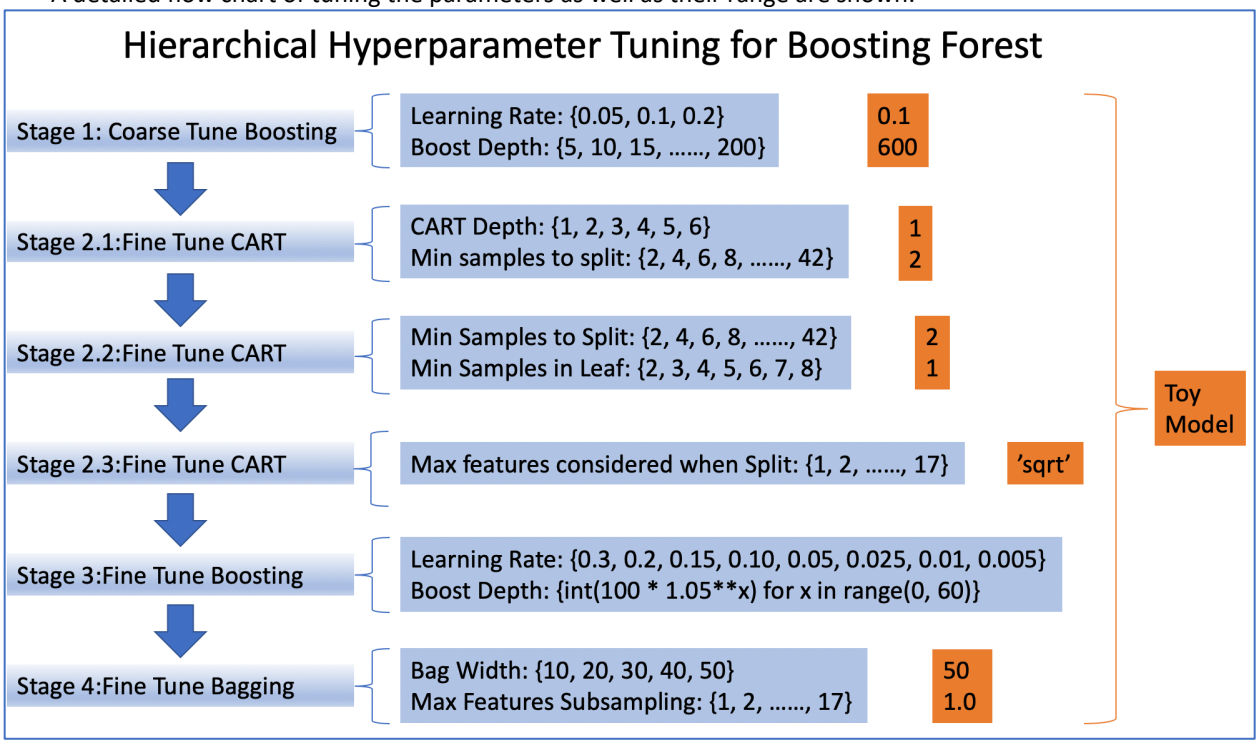
networks. Thus, we are not activating 'early stopping tricks' provided by Sklearn's AdaBoost API. Also, subsampling data points are un-abled in the bagging part, and we want every individual boosting tree to be fed with the full-size dataset because we are already suffering from relatively insufficient data.

Aside from those that will never change during the model selection process mentioned above, there are other parameters that needs to be tuned, however will not fit into grid search loops. Since subsampling features are abled when bagging, whether to bootstrap is a binary choice and is an example. Also, the kind of loss we're using for growing boosting trees and criterion used for splitting in CART are both such hyperparameters.

Those left are now all left to the hierarchical grid search process, they include:

CART Parameters: max depth, max features, min samples split & min samples leaf
Boost Parameters: boost depth, subsample ratio, learning rate
Bagging Parameters: bagging width, max features

A detailed flow chart of tuning the parameters as well as their range are shown:



A toy model configured with heuristics are executed first, and the values of parameters are listed above in the orange boxes. We can see that it did nothing to prevent overfitting and extremely biased splitting. The results are:

```
Performing untuned BoostingForest using initial parameters
CV Performance of Untuned BoostingForest:
[0.967032967032967, 0.9560439560439561, 0.989010989010989, 0.9438202247191011, 0.9775280898876404]
Mean_Acc 0.9667, Std_Acc 0.0177
```

Untuned Boosting Forest performs slightly worse than baseline model in average but has lower standard deviation across folds. We are generally expecting an improvement after fine tuning the model.

The detailed fine-tuning process are as follows. First coarse tune the structure of the boosting tree. In stage 1, learning rate should be rather high, and boost depth should be rather shallow. Normally, it will increase the risk of overfitting, but it will also lead to a 'weaker' CART as wished in later stage. And also, such configuration can also speed up the training and cross validation process.

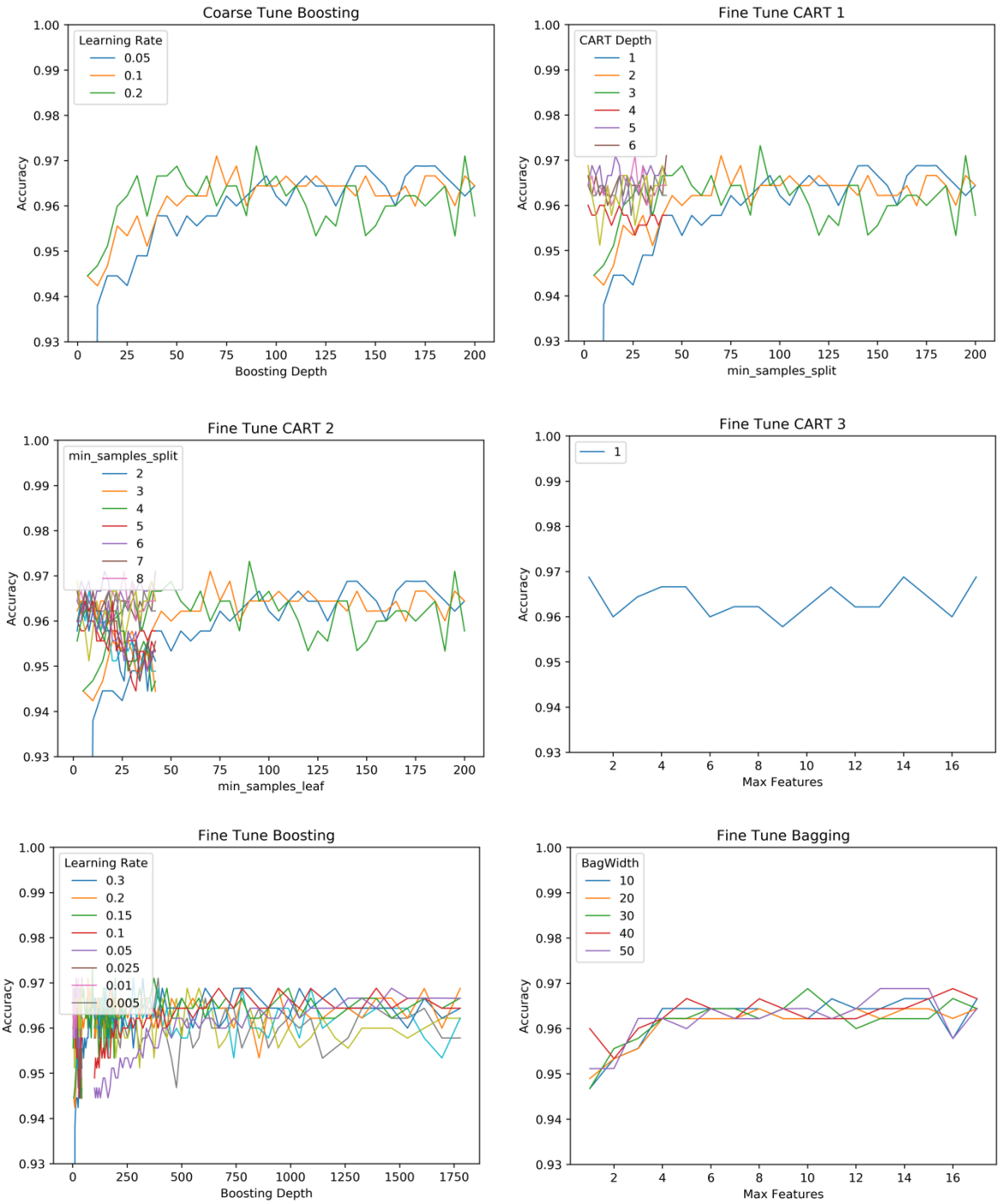
For stage 2, the order is determined by the impact each parameter has on the outcome. Those with larger impact should come first. And also, they are also grouped by their impact types. For example, parameters in stage 2.1, 2.2 goes before ***'Max feature considered when splitting'*** because they control the complexity of the model, which is related to underfit/overfit. And splitting conditions in stage 2.2 also controls the biasness of each spilt, i.e. we are not allowing to split the region if one of the two has substantially lesser data points than the other one. Stage 2.3 are to determine the best variance between individual CARTs by tuning how many features each CART is facing. It has the smallest impact.

In stage 3, we lower the learning rate/increase the boosting depth proportionally by a factor of 1.05, to make the Boosting tree model more stable, and to see if we can squeeze more accuracy out of the model.

$$\boldsymbol{BoostDepth = 100 \times 1.05^{step}\ for\ step = 0\sim60}$$

In stage 4, a bagging wrapper method is implemented on top of the Boosting tree just tuned, and again perform grid search to find the best forest width and subsamples to feed to each boosting tree.

The model selection results are show below:



Worth noticing that, the six plots are not generated in one shot. They are in pair sequentially tuned, and their best are chosen by hand before entering the next stage. It can be clearly shown that tree-based models exploit a pattern of "reaching limits", and the value of hyperparameters doesn't quite matter much except edge cases. The curves oscillate

hardly with the parameter values without having an overall tendency of going up/down, such cases which may be dominated by the randomness of train/valid splits in cross-validation rather than the model itself. In order to carry on choosing one to go into the next stage, the criterion is set to be printing the hyperparameter values that has a cross validation accuracy above a threshold (96.5% in this case), and take the mean of the vicinity of hyperparameters as the best one with highest frequency of getting above that threshold. The final configuration of the parameters tuned are the following. Worth mentioning that using "Exponential" loss actually recovers AdaBoost algorithm:

$$\boldsymbol{Boosting}: \boldsymbol{Learning\ rate} = \mathbf{0.025}, \boldsymbol{Depth} = \mathbf{400}, \boldsymbol{Loss} = '\boldsymbol{exponential}', \boldsymbol{critirion} = '\boldsymbol{friedman\_mse}'$$
$$\boldsymbol{CART}: \boldsymbol{Depth} = \mathbf{5}, \boldsymbol{MinSamplesLeaf} = \mathbf{6}, \boldsymbol{MinSamplesSplit} = \mathbf{10}$$
$$\boldsymbol{Bagging}: \boldsymbol{Width} = \mathbf{50}, \boldsymbol{MaxFeature} = \mathbf{15}, \boldsymbol{BootStrap} = \boldsymbol{False}$$

The final CV results of the AdaBoost Tree & Boosting Forest are the following:

```
Final CV Performance of BoostingTree:
Mean_Acc 0.9644, Std_Acc 0.0167
 [0.967032967032967, 0.9560439560439561, 0.989010989010989, 0.9438202247191011, 0.9662921348314607]
Final CV Performance of BoostingForest:
Mean_Acc 0.9666, Std_Acc 0.0238
 [0.967032967032967, 0.967032967032967, 1.0, 0.9325842696629213, 0.9662921348314607]
```

(c): Comments from the training process:

1. Tree based models tuned according to the process above turned out having the same level performance with baseline models.

2. The performance gain of bagging over individual boosting trees is not that distinguishable from single boosting tree neither. AdaBoost even have a lower variance cross all 5-folds than Boosting Forest.

3. The performance between "Toy" & "Fine" models are also not distinguishable neither.

3.4.3 Linear model: Elastic net logistic regression[8]:

(a) Models & Method Description:

As a typical linear method, logistic regression is a discriminative method that finds the best decision boundary by maximizing the likelihood function. It inherits the merits of low capacity as a linear model, which makes it rather stable compared with tree-based methods and harder to overfit. It is also compatible with numerous statistical regularization techniques such as L1 & L2, aka LASSO & RIDGE, further decreasing the risk of overfitting. L1 & L2 penalization terms have their own advantages, such as L1 creates sparse models that is expected to be more robust to irrelevant features and L2's convexity in some extent helps the convergence of algorithms like gradient descent. Here we implemented Elastic Net logistic regression, with the penalty term being a linear combination of L1 & L2:

$$\boldsymbol{TargetFunction} = \frac{\mathbf{1}}{|\mathcal{D}|}\sum_{\mathcal{D}} \boldsymbol{L}(\boldsymbol{x}; \vec{\boldsymbol{w}}) + \boldsymbol{\lambda\rho}\|\vec{\boldsymbol{w}}\|_{\mathbf{1}} + \boldsymbol{\lambda}(\mathbf{1}-\boldsymbol{\rho})\|\vec{\boldsymbol{w}}\|_{\mathbf{2}}(\mathbf{0} \leq \boldsymbol{\rho} \leq \mathbf{1})$$

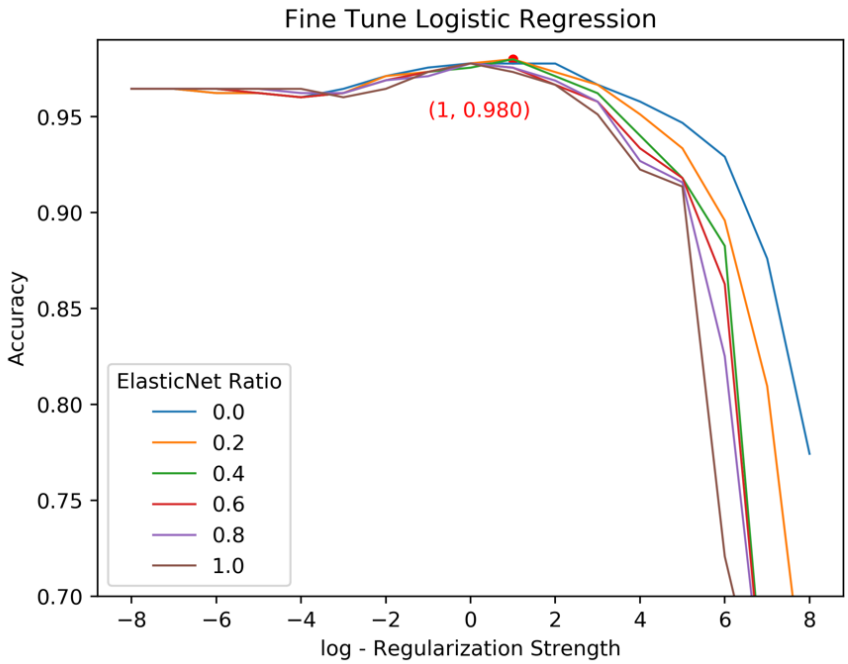
(b) Training Process & Model Selection:

L1 & L2 are recovered in the Elastic Net regularization by setting $\rho = 1, 0$ respectively. The best performing hyperparameter pair (ρ, λ) can be determined by grid search, and the search range are as follows:

$$\boldsymbol{Elastic\ Net\ Ratio}\ \boldsymbol{\rho}: \{\mathbf{0}, \mathbf{0.2}, \mathbf{0.4}, \mathbf{0.6}, \mathbf{0.8}, \mathbf{1.0}\}$$
$$\boldsymbol{Regularization\ Strength}\ \boldsymbol{\lambda}\ (\boldsymbol{in\ log\ scale}): \{-\mathbf{8}, -\mathbf{7}, -\mathbf{6}, \ldots\ldots, \mathbf{7}, \mathbf{8}\}$$

The model selection results are as follows:

[8] Library used: sklearn.linear_model.LogisticRegression(), generated by models.AccSqueeze_linear_main(), accSqueeze_linear_last()



The model selection results from cross-validation is much better than that from tree-based models above. With gentle, almost none, regularization strength, models with different Elastic Net ratio begins from the same starting point. As the strength increases, all the models exploited "rise and fall" pattern, which is a standard performance curve we are expecting from hyperparameter tuning. It's also interesting to see that, when we reached a big enough regularization strength (>exp(2)), the performance grows with the decrease of elastic net ratio.

We choose $\big(\boldsymbol{\rho}, \boldsymbol{log}(\boldsymbol{\lambda})\big) = (\mathbf{0.2}, \mathbf{1})$ to be the best hyperparameter pair from the chart above. Notice that $\big(\boldsymbol{\rho}, \boldsymbol{log}(\boldsymbol{\lambda})\big) = (\mathbf{0.4}, \mathbf{1})$ also achieved average CV accuracy 98% but accuracy in its regularization's vicinity is lower than that of $\big(\boldsymbol{\rho}, \boldsymbol{log}(\boldsymbol{\lambda})\big) = (\mathbf{0.2}, \mathbf{1})$, thus considered as less stable.

The final CV results of Elastic Net logistic regression are the following:

```
Final CV Performance of ElasticNet Logistic Regression:
[0.978021978021978, 0.978021978021978, 1.0, 0.9662921348314607, 0.9775280898876404]
Mean_Acc 0.9800, Std_Acc 0.0123
```

(c): Observations from the training process:

Elastic net logistic regression has a better performance than tree-based models.

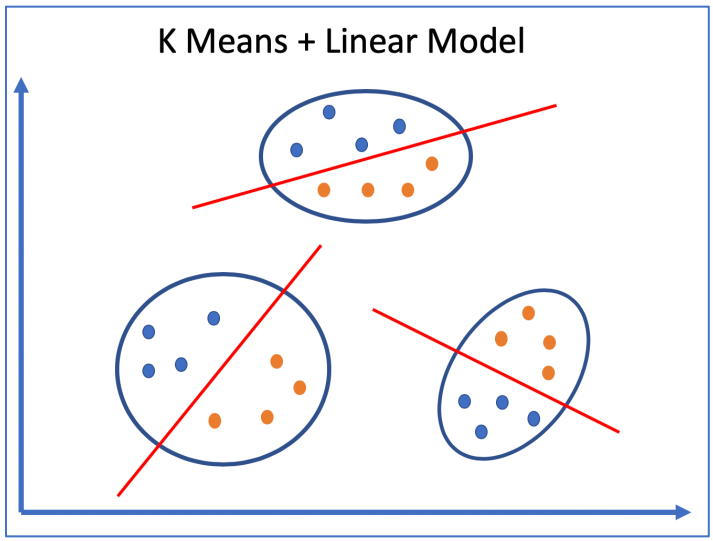
3.4.4 Combined model: K means clustering + logistic regression[9]:

(a) Models & Method Description:

Among "tricks" of improving the performance of linear models, one attempt is to utilize unsupervised method to cluster the data points as a "preprocessing" method, and then implement logistic regression in each of one these clusters. Such wrapper methods[10] cover the blind spot of base linear models in cases such as where the true underlying model being a mixture model and no linear boundary exists. One can argue that such combinatorial approach is expected to be at least as good as direct fitting one single logistic regression model, but in reality, and also can be seen later in this project, the performance may be restricted by the prediction quality from the upstream unsupervised method. An illustration is shown below:

[9] Library used: sklearn.linear_model.LogisticRegression(), cluster.KMeans(),generated by models.AccSqueeze_kMeans_main(), AccSqueeze_kMeans_last()

[10] the process is self-implemented with the help of previous libraries



(b) Training Process & Model Selection:

Here we use K means clustering as the upstream clustering algorithm, and the implementation of such approach calls for 3 hyperparameters:

$$\boldsymbol{Number\ of\ Clusters}: \boldsymbol{K}, \boldsymbol{Penalty\ Strengh}: \boldsymbol{\lambda}, \boldsymbol{Elastic\ Net\ Ratio}: \boldsymbol{\rho}$$

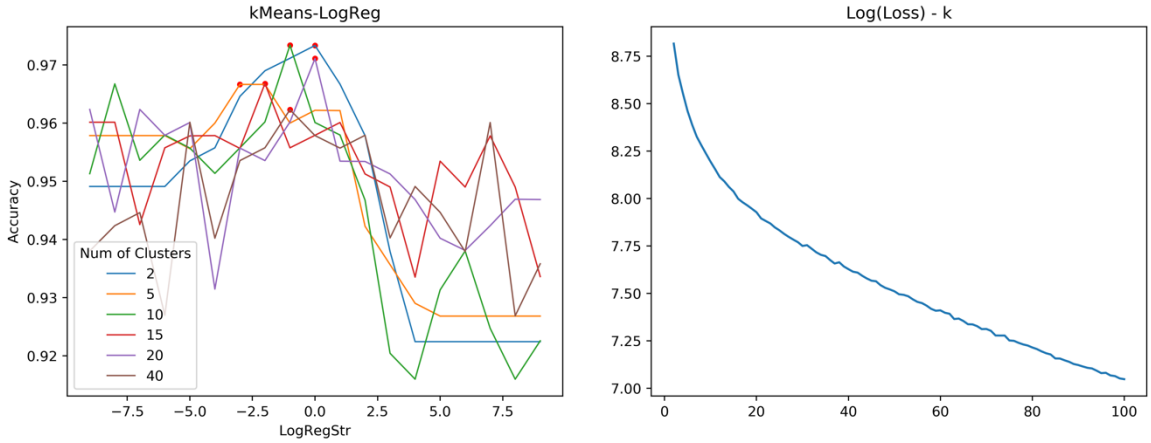
Here, we maintain the best performing ρ from the previous section where $\rho = 0.2$. And for the number of clusters, we need to shrink the range to cross validate from, because large k will definitely result in overfit and also result in meaningless increase of time consumption. Though we cannot fully falsify the possibility of a large such model with hundreds of clusters having a good performance, we still seek to shrink the range of k before cross-validation. We first explored the relationship between k and minimum loss achieved by k means, with k ranging from 2 to 100 and loss being the standard K Means loss:

$$\mathcal{L} = \sum_{c \in \{c_1, \ldots\ldots, c_k\}} \frac{1}{|c|} \sum_{x_i \in c} \|x_i - \overline{\mu_c}\|_2^2$$

We observed a knee point on the result, and take that as the upper bound of the cross-validation range for k. It's just a method arose from heuristics and can be interpreted as marginal performance gain of clustering has a sharp decrease beyond the knee point. The final grid search range for both hyperparameters are the following:

$$\boldsymbol{Number\ of\ Cluster\ k}: \{\mathbf{2}, \mathbf{5}, \mathbf{10}, \mathbf{15}, \mathbf{20}, \mathbf{40}\}$$
$$\boldsymbol{Regularization\ Strength\ \lambda\ (in\ log\ scale)}: \{-\mathbf{8}, -\mathbf{7}, -\mathbf{6}, \ldots\ldots, \mathbf{7}, \mathbf{8}\}$$

The model selection results are as follows:



One interesting observation from the left figure is that large regularization strength works better with more clusters. It can be partially explained as the more clusters model has, patterns within each cluster tends to be simpler, thus simpler models have better performance. This argument should be more obvious if we had smaller steps in regularization strength. We choose $(\boldsymbol{k}, \boldsymbol{log}(\boldsymbol{\lambda})) = (\mathbf{2}, \mathbf{0})$ to be the best hyperparameter pair from the chart above. Notice that $(\boldsymbol{k}, \boldsymbol{log}(\boldsymbol{\lambda})) = (\mathbf{10}, -\mathbf{1})$ also achieved the same average CV accuracy as what we choose, but as the same argument in the last section, the vicinity of its regularization strength is smaller than our choice, thus discarded.

The final CV results of k means + logistic regression is the following:

```
Final CV Performance of kMeans Preprocessed logRegresion:
[0.978021978021978, 0.967032967032967, 0.989010989010989, 0.9550561797752809, 0.9775280898876404]
Mean_Acc 0.9733, Std_Acc 0.0128
```

(c): Observations from the training process:

Such combined approach doesn't further boost the performance of linear models.

3.5 Exploration Part:

3.5.1 Semi-supervised method: l1-SSSVM[11]:

Self-training, aka Yarowsky Algorithm [7][8], is simple yet useful approach when we want to squeeze information, mainly by the distribution, from unlabeled data when facing a semi-supervised problem. It's compatible with most algorithm/model which possesses the notion of confidence, such as label purity in KNN or distance to the hyperplane/decision boundary in linear models. Whether "kernel machines" also have such measurements of confidence remains to be further explored. Here we used SVM as the base classifier to perform self-learning on partially, artificially masked data points. As one can argue, self-training methods are extremely vulnerable to early mistakes, because such mistakes could reinforce itself which greatly impacts predictions later on. There are more sophisticated semi-supervised learning methods that could consider such problem, e.g. co-training, or 'tricks', like allows 'un-labeling' unconfident labeled data. But it can be shown from our results that early mistakes are actually not that devastating, and models still have the chance to correct itself later on.

The main idea for Yarowsky Algorithm is to utilize the any measure of confidence – iteratively cast predictions on the unlabeled dataset, take the most, or a few, confident unlabeled data point(s), and take their predicted labels as true labels. In l1-SVM implemented here, the relative/normalized distance is considered as confidence – the larger the distance to the decision boundary, the more confident current model is with respect to the prediction of a certain data point. Distance can be thankfully retrieved by Sklearn's API.[12]

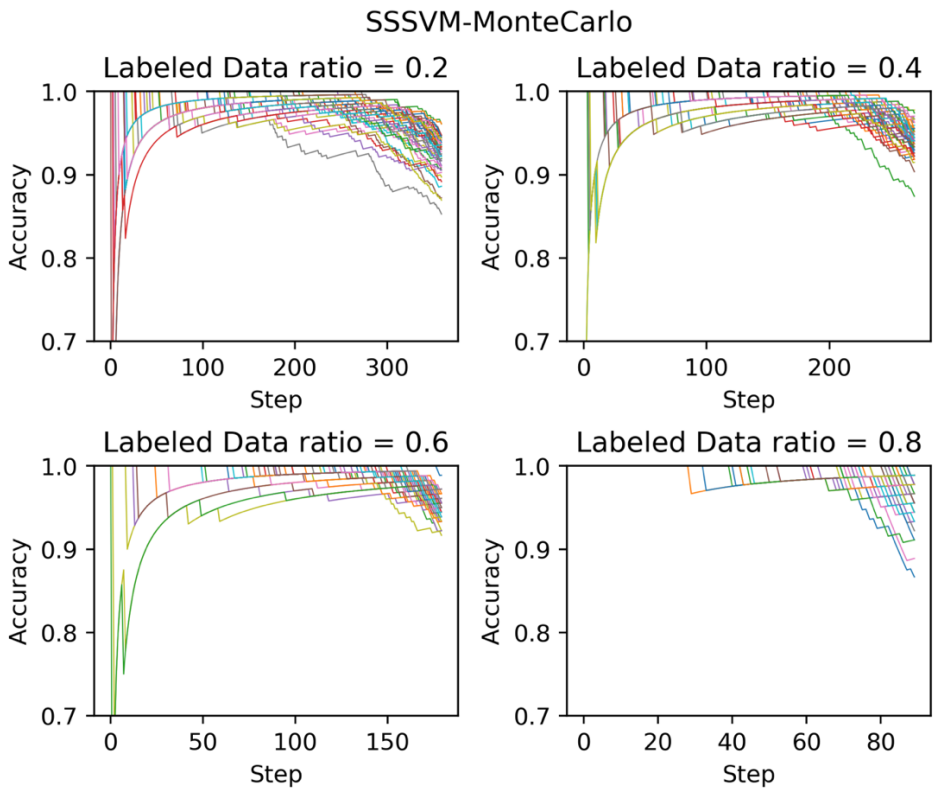
In our particular setting of this project, regularization strength is first cross validated on the labeled data and fixed for this single execution. Then self-training process begins, and iteratively adds the most confident data point into the labeled dataset until no unlabeled data points are left. One-time traverse of this process is considered as one single execution, and they are repeated in a Monte-Carlo manner with different randomly sampled labeled datasets 50 times ("Starting Datasets"). And finally, all the above process is repeated for different labeled/unlabeled dataset size ratio[13].

The results are shown below:

[11] Library used: sklearn.svm.LinearSVC(), sklearn.model_selection.StratifiedKFold(), generated by models.Explore_SSSVM_main()

[12] By the .decision_function() method of the LinearSVC class.

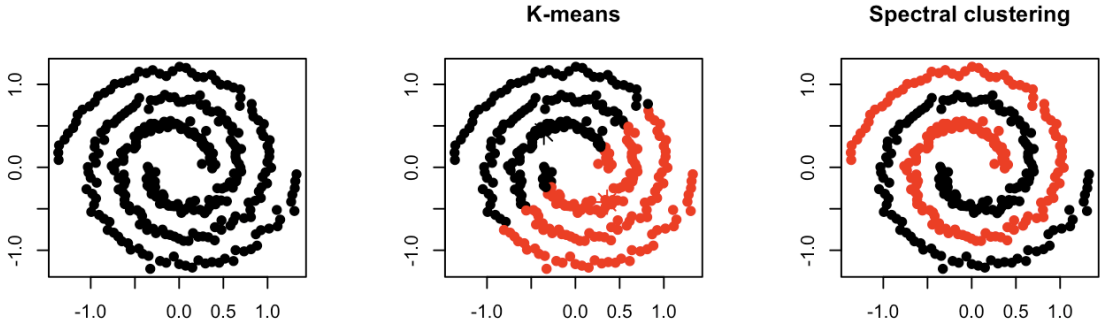
[13] the process is self-implemented with the help of previous libraries




```
Implementing Semi-supervised Learning: l1 - SVM
Ratio = 0.20 Completed, Time Elapsed: 0:00:55.607242
Ratio = 0.40 Completed, Time Elapsed: 0:03:04.643691
Ratio = 0.60 Completed, Time Elapsed: 0:11:41.080383
Ratio = 0.80 Completed, Time Elapsed: 0:07:10.142969
Final Statistics for labeled ratio: 0.2, 0.4, 0.6, 0.8
Ratio = 0.2, Mean = 0.9239, Std = 0.0222
Ratio = 0.4, Mean = 0.9410, Std = 0.0189
Ratio = 0.6, Mean = 0.9509, Std = 0.0168
Ratio = 0.8, Mean = 0.9629, Std = 0.0284
```

3.5.2 Unsupervised method: Spectral Clustering[14]:

K means clustering, as an unsupervised method depending on compactness over the center of the clusters, usually fails because it runs on a strong assumption that data points assigned to a cluster are spherical about the cluster center. With quite the same underlying thought of applying 'kernel tricks' to SVM, spectral clustering, as an alternative to K Means, works on pairwise metrics between the data points, instead of those to pre-assumed "cluster means". Moreover, with the notion of 'metric', Spectral Clustering can be easily extended to all mathematically well-defined metrics functions, resulting in its great adaptability to all kinds of underlying true generating models. On the other hand, Spectral Clustering suffers from exploding time complexity w.r.t. number of data points, for either constructing Laplacian matrix L or performing spectral analysis. And it still requires executing k means, after constructing the Graph Laplacian matrix [9].



[14] Library used: sklearn.cluster.SpectralClustering, generated by models.Explore_SpeCluster_main(), Explore_SpeCluster_last()

Spectral Clustering algorithm can be briefly stated as follows:

1. Evaluate all pairwise similarities between data points. The similarities covered in this project includes:

$$\boldsymbol{Gaussian}(x_i, x_j; \gamma) = exp\left(-\gamma \|x_i - x_j\|_2^2\right)$$
$$\boldsymbol{Laplacian}(x_i, x_j; \gamma) = exp\left(-\gamma \|x_i - x_j\|_1^2\right)$$
$$\boldsymbol{Polynomial}(x_i, x_j; \gamma, d) = \left(\gamma < x_i, x_j > + c_0\right)^d$$
$$\boldsymbol{NearestNeighbor}(x_i, x_j) = \mathbf{1}\ if\ x_i\ is\ k - neighbors\ of\ x_j\ and\ also\ vice\ versa$$

2. Construct Laplacian matrix by the similarities computed above:

$$L_{ij} = \begin{cases} \sum_{\{j|(i,j)\in E\}} s_{ij} & if\ i = j \\ -s_{ij} & if\ i \neq j \\ 0 & if\ (i,j) \notin E \end{cases}$$

3. Perform spectral decomposition and projecting data points:

Notice that L by its construction will be a rank deficient, symmetric matrix. Thus, its eigenvalues can be sorted as:

$$0 = \lambda_1 \leq \lambda_2 \leq \lambda_3 \leq \lambda_4 \leq \ldots\ldots \leq \lambda_N$$

With pre-determined dimension of the projected feature space, say k, we stack eigenvectors corresponding to k-smallest eigenvalues other than 0, and form projection matrix V:

$$V^{N \times k} = [v_2\ v_3\ v_4\ v_5\ \ldots\ldots v_{k+1}]$$

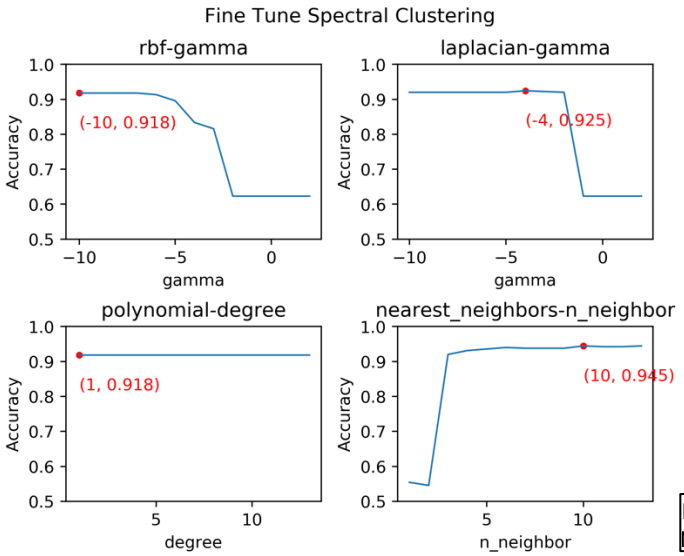
4. Generate the projected data points, and perform K means clustering:

$$\widehat{x_k} = \overrightarrow{V_{k\cdot}}\ (kth\ row\ of\ V)$$

Unsupervised methods can be easily modified to fit into a problem with a supervised goal. We can cast a majority vote in each cluster and assign that majority label to any future data that goes into this cluster. But as expected, pure unsupervised approach generally does worse than a supervised learning model/algorithm because it doesn't make use of the information labels provides.

(b) Training Process & Model Selection:

Grid search cross-validation is performed over the 4 metrics mentioned above, and with their own parameters. The results can be generated by and are shown below:




```
Final Performance of Spectral Clustering:
Mean_Acc 0.9446
```

4. Final Test Results and Interpretation:

The final results of all the models tested are organized in the chart below:[15]

	KNN		AdaBoost		BoostingForest		ElasticNetLR		KMeans+LR		SSSVM		SpeClustering	
Train_Mean	0.9688		0.9644		0.9666		0.9800		0.9733		NaN		0.9446	
Train_Std	0.0217		0.0167		0.0238		0.0123		0.0128		NaN		NaN	
Test_Acc	0.9664		0.9496		0.9496		0.9748		0.9664		0.8571		0.9402	
Test_Precision	0.9130		0.8913		0.8913		0.9438		0.9130		0.9348		0.8443	
Test_Recall	1.0000		0.9762		0.9762		1.0000		1.0000		0.7544		0.9944	
Test_F1	0.9545		0.9318		0.9318		0.9663		0.9545		0.8350		0.9133	
Test_ ConfusionMX	73	4	72	5	72	5	73	3	73	4	59	3	356	33
	0	42	1	41	1	41	0	43	0	42	14	43	1	179

Comment:

1. The mean/std of cross-validation are just copies from previous sections, for comparison.
2. Test results for SSSVM are transductive and are generated by final model that start from single generation of masked/labeled dataset with ratio 1:1. I.e., the model doesn't learn anything from the test data. There are not final train performances for ratio 0.5 above, thus plugged in "NaN".
3. Since Spectral Clustering are unsupervised methods, the test results are generated from full train + test data.

Up to now, conclusions can be made on the "Breast Cancer Dataset" & models used. This dataset is relative well behaved, and the models tried all gets a pretty good performance (accuracy) on the binary classification risen. This is consistent with the information that "176 consecutive new patients are correctly diagnosed as of Nov.1995" provided by [6]. Also, the best estimated 10-fold CV accuracy reported by [6] is 97.5%, which is slightly lower than our best "ElasticNetLR", but still on the same level. Such well-behavior of the dataset is bad news for models with large, even potentially infinite, capacity like Adaboost/Random Forest. Data points beside the "decision boundary", in a broad sense, may be dominated by noises and leaves the last 2 ~ 3 percent accuracy hard, even impossible to squeeze.

This project is a perfect example of the "No free lunch" theorem. Carefully tuned, large capacity models are being outperformed by linear models. Such may result from "deterministic noise", or is just the true distribution is just unbelievably to be "linear" and "stochastic noise" are to blame for the errors. This is determined by the nature of the dataset and its underlying true probability distribution, and we can only accept as it is.

Another observation came from the training procedure of the tree-based model. The role hyperparameters, or one single hyperparameter, play in tree-based models are very limited. In [5], AdaBoost gained only <3% accuracy after such tedious tuning process in another classification task. Apart from the task itself being simple in this project, it may lead to an elementary argument that tree-based ensemble methods are not that sensitive to settings of hyperparameters like other model like logistic regression or SVM. There are hundreds of CARTs in one boosting structure, and it doesn't quite matter much whether CART depth is limited to 3 or 4, or does CART subsample 3 or 4 features when splitting. But that doesn't wipe away the necessity of fine-tuning Boosting/Bagging parameters, because it's still case/dataset dependent and "knowledge comes from practice". Following the heuristics by keeping the model capacity of individual CART low and having a relative shallow boosting tree will always be a good "Toy" model to start with when fine tuning such models.

From the second part of the project, semi-supervised/unsupervised methods are worse than supervised methods as expected. What also is consistent with intuition is that SSSVM's inductive accuracy (in-training accuracy) goes up with the size of "starting dataset". And the reason why spectral clustering achieved better accuracy than SSSVM is that the kernel "Nearest Neighbor" along with cluster number 2 makes it rather a K-Nearest neighbor "classifier" than a clustering algorithm. Dataset's linear separability is supported by the success of logistic regression models and also explains the unexpected performance of Spectral Clustering with 2 clusters. "Test" results for semi-supervised/unsupervised models are in theory not directly comparable to supervised methods, because in reality, such problems do not have true labels to generate "Test" results. They have their own measurements of performance. Thus, they are just considered in the exploration part of this project to exploit the information/contribution labels provide in a supervised classification problem.

The fact that precision is substantially lower than recall shows the fact that models tend to miss "malignant" patients (those with breast cancer) more. There are two main reasons for this:

1. Class "malignant" is minority class in the dataset, taking 37.26% of the total dataset. Though it can hardly be considered as severe cases of class imbalance, it still contributes to the observation above.
2. The natural background of this problem is "diseases diagnosis" and detecting all the positive cases has long been a huge task that applies to all diseases, never to mention that we are only predicting upon the imaging of cell nuclei, letting out other clinical evidence such as symptoms.

[15] Generated by models.XXX_XXX_test(), or main.showTestResult()

5. Contribution of each team member:

This project as well as its report are done and composed entirely by Chengyao Wang.

6. Conclusions & Further Expectations:

This project achieved a highest accuracy with elastic net logistic regression at an CV/test accuracy of 98%/97.48%, slightly higher than the CV accuracy reported in [6]. Other supervised models exploited a similar, however worse performance.

Also, semi-supervised & unsupervised methods are also implemented in this supervised classification schema, but as expected have a worse performance than supervised methods.

Further expectations of this project include:

1. Measurement of performance change from accuracy to recall.
2. "Soften" the prediction of K means and cast weighted vote among all base logistic regression classifiers.
3. Survey for more sophisticated, theory guided tree-based ensemble model tuning strategy.
4. Apply the models above to a more difficult problem to see if conclusion above still holds.

7. References:

[1] K. P. Bennett, "Decision Tree Construction Via Linear Programming." Proceedings of the 4th Midwest Artificial Intelligence and Cognitive Science Society, pp. 97-101, 1992

[2] K. P. Bennett and O. L. Mangasarian: "Robust Linear Programming Discrimination of Two Linearly Inseparable Sets", Optimization Methods and Software 1, 1992, 23-34.

[3] Wikipedia, Coefficient of Variation, https://en.wikipedia.org/wiki/Coefficient_of_variation.

[4] Max Welling, Fisher Linear Discriminant Analysis, https://www.ics.uci.edu/~welling/teaching/273ASpring09/Fisher-LDA.pdf

[5] Aarshay Jain, "Complete Machine Learning Guide to Parameter Tuning in Gradient Boosting (GBM) in Python", https://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python/

[6] Readme of the dataset.

[7] Wikipedia, Yarowsky, https://en.wikipedia.org/wiki/Yarowsky_algorithm

[8] GholamReza Haffari and Anoop Sarkar, "Analysis of Semi-Supervised Learning with the Yarowsky Algorithm", https://arxiv.org/pdf/1206.5240.pdf

[9] Neerja Doshi, Spectral Clustering, https://towardsdatascience.com/spectral-clustering-82d3cff3d3b7

8. Appendix:

Dataset Information:

"Breast Cancer Wisconsin (Diagnostic) Data Set"

URL: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29

Code organization & test result generation:

Main.py:	Highest level script for generating results.
dataGen.py:	Class "dataGenerator" for data reading, preprocessing, visualization & data fetching.
modules.py:	Class "models" containing all training & testing process of models used in this project.
utils.py:	Class "util" for all graph plotting, dataset generating/fetching for models, result printing.

Dependencies:

Python:	3.7.4
Scikit-learn:	0.21.3
Pandas:	0.25.1
Seaborn:	0.9.0
Other:	numpy, datetime, shutil, os, statistics, matplotlib.pyplot, random

To generate the test results, change to the directory & use command:

```
python main.py
```