통계적 머신러닝을 통한 Kannada Digits 분류

2014150001 통계학과 이승훈

본 프로젝트를 통해 숫자 이미지를 머신러닝 알고리듬을 통해 학습시켜 숫자 분류를 하고자 한다. 먼저 차원 축소를 통해 더 적은 수의 변수로 데이터의 많은 부분을 설명할 수 있도록 feature extraction을 하고자 한다. 또한, 각 모델에 대한 분류 정확도와 계산시간, 그리고 과적합 위험 정도에 초점을 맞추고 숫자 이미지 분류를 하는데 있어 통계적 머신러닝 모델을 비교할 것이다.

1. 데이터 준비

1-1 데이터에 대한 설명

일상적으로 익숙한 아라비아 숫자 분류를 할 수도 있지만, 좀 더 생소한 Kannada digits 를 분류해 보기로 한다. Kannada는 인도 서부의 Karnataka에 거주하는 사람들이 주로 사용하는 언어이며, 숫자 표기가 일상적으로 쓰이는 표기와 많이 다르기 때문에 좀 더 흥미로울 것이라고 생각했다. 데이터는 케글의 Kannada MNIST competition (https://www.kaggle.com/c/Kannada-MNIST)에서의 데이터셋에서 찾을 수 있었다. 데이터의 형태를 다음과 같이 간략하게 살펴 볼 수 있다.

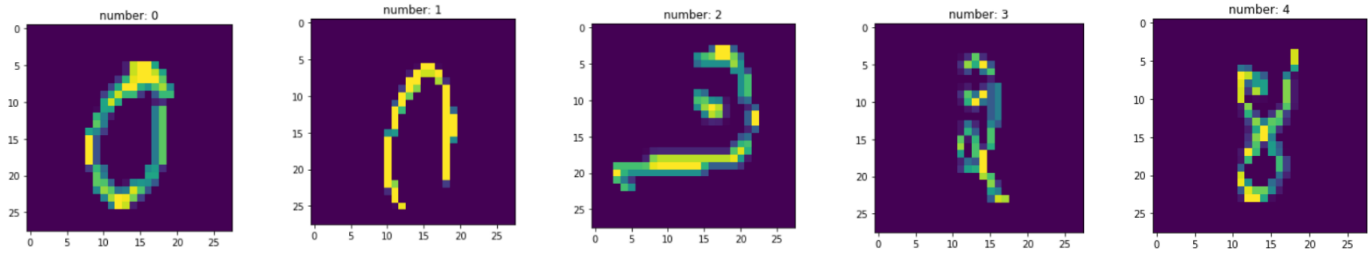
```
data.head()
```

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
2	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
3	3	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
4	4	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0

5 rows × 785 columns

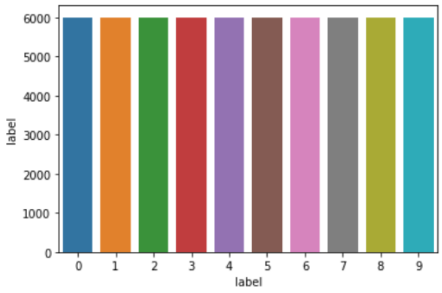
데이터는 row 가 60000개, column이 785개였다. 한 개의 목적(target) 변수와 784개의 특성(feature) 변수 로 이루어져 있었으며, 각 feature variable 들이 28*28 = 784 사이즈의 픽셀들을 나타낸다는 것을 알 수 있었다. feature variable의 경우 0에서 255 까지의 값을 가지고 있었다.

다음 그림은 Kannada digits의 예시들이다. 차례로 0부터 4까지 나열했다.



Target variable(label) 의 경우 0에서 9까지의 값을 가지고 있었으며 각각 모두 6000개의 관측값을 균일하게 가지고 있었다. 원 핫 엔코딩을 하고, 훈련 데이터와 테스트 데이터를 7: 3 으로 나누어 주었다. (scaleling 적용)

Target variable 빈도

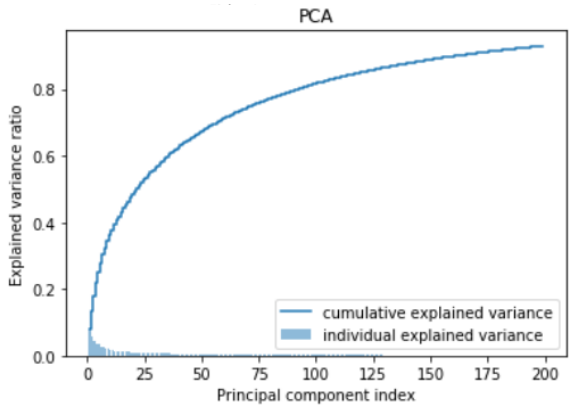


1-2 차원 축소문제

Kannada 데이터는 총 784개의 픽셀(특성변수)을 가지고 있으며, 이 모든 특성변수를 숫자를 분류하는 정보로서 이용하는 것은 불필요할 뿐더러 오히려 통계적 머신러닝 알고리듬의 성능을 낮출 수도 있다. 따라서 특성변수 대부분의 정보를 가진 새로운 특성변수를 추출하여 차원을 축소할 필요가 있다.

1-2-1. Principal Component Analysis

먼저 PCA를 통한 차원축소를 위해 아이겐밸류를 구하고, 주성분들이 데이터의 전체 분산을 얼마만큼 설명할 수 있는지 알아보았으며, 결과는 다음과 같았다.

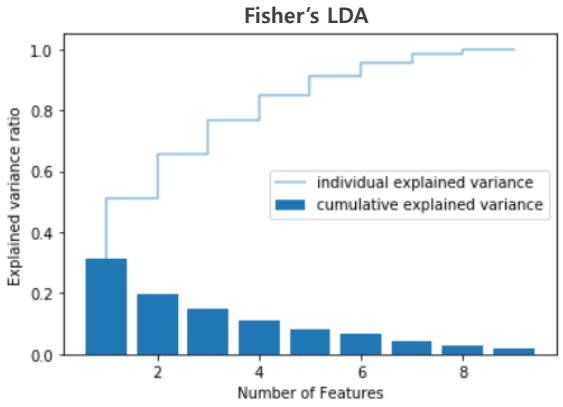


변수의 개수가 많은 만큼 첫 10개 정도의 주성분을 제외하고는, 주성분들이 데이터의 전체 분산을 설명할 수 있는 정도가 매우 약했다. 첫 주성분 50개 까지는 사용해야 데이터 분산의 60% 이상 설명할 수 있었으며, 100 개에 가깝게 사용하더라도 전체 데이터 분산의 80%도 설명할 수 없었다.

첨언) Kernelized PCA를 쓸 수도 있겠지만, 이 데이터에 대해서는 메모리 문제로 인해 비효율적이라 느껴 쓰지 않았다.

1-2-2. Fisher's Linear Discriminant Analysis

Fisher's LDA의 경우에도 마찬가지로 새로운 특성변수를 생성하여, 이들이 전체 데이터의 얼마만큼 설명할 수 있는지 알아보았다. (9 차원을 형성함)



PCA를 통한 차원축소와는 달리, Fisher's LDA 차원축소를 통해 생성된 새로운 변수들은 전체 데이터 정보의 많은 부분을 설명할 수 있는 수준이었다. 첫 번째 변수만으로 31.1% 가량을 설명했으며, 첫 6개의 변수를 사용할 경우 91.4% 나 설명이 가능했다. 이는 Fisher's LDA는 PCA 와는 달리 y의 정보를 이용하여 클래스를 잘 분류하는 벡터를 찾기 때문인듯 하다. 다음 표는 동일한 옵션에서 PCA와 Fisher's LDA를 통해 얻은 새로운 변수로 몇몇 모델을 적합 시켰을 때 얻은 결과이다. (본 보고서에서의 계산시간은 훈련 셋을 통해 모델을 적합 시키고, 트레인 데이터와 테스트 데이터를 통해 예측까지 완료하는데 걸린 시간을 포함한다)

	PCA(n_components = 25)		LDA(n_components=6)	
	Train Accuracy / Test Accuracy	계산시간	Train Accuracy / Test Accuracy	계산시간
KNN	0.984/0.975	125.236	0.958/0.939	7.393
Logistic Reg	0.930/0.924	20.877	0.931/0.925	2.444
SVM	0.959/0.955	10.194	0.944/0.937	6.418
Classification Tree	0.799/0.792	0.017	0.960/0.924	0.013

첨언) 계산속도를 높이기 위해 10개의 주성분만 사용할 경우, 대부분의 모델에서 90% 미만의 accuracy가 나왔기 때문에 accuracy를 어느정도 높일 수 있도록 주성분 개수를 25개로 잡았다.

모델을 적합 시키는데 포함된 변수의 수에 차이가 많음에도 불구하고 (PCA의 주성분 수를 크게 잡음), 모든 경우에서 accuracy의 차이는 크지 않았으며, Fisher's LDA를 통해 만들어진 변수를 통해 모델을 적합 시키고, 클래스를 분류했을 때 계산 속도가 PCA의 경우보다 월등히 빨랐다. 따라서 Kannada 데이터 셋과 같이 특성변수의 수가

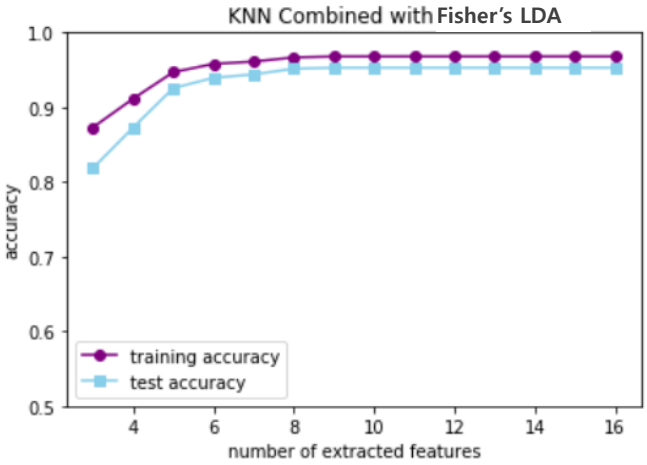
많은 상황에서(또는 target에 대한 정보가 있는 상황에서) 계산속도를 빠르게 하는 것이 목표라면 Fisher's LDA를 사용하여 차원축소 하는 것이 적합해 보인다. 하지만 다소 시간이 걸리더라도 PCA를 통해 차원축소를 하여, 특성변수를 많이 사용하는 것이 더 적합한 경우 역시 있었다. (KNN의 경우가 그 예이다.)

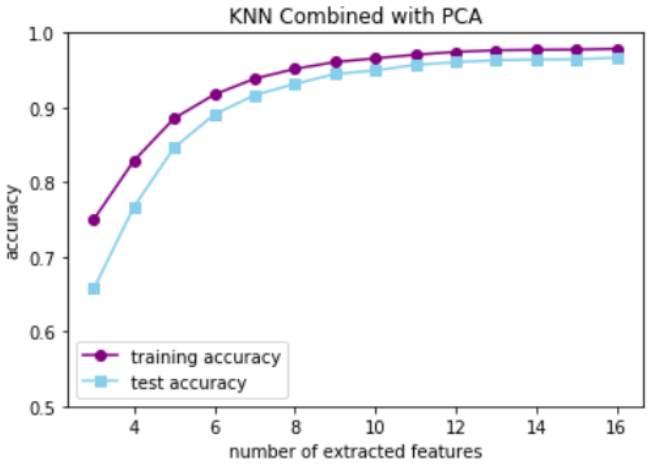
분류 정확도도 중요하지만 계산 시간 역시 포기할 수 없는 부분이라고 생각했다. 따라서 본 프로젝트에서는 분류 정확도를 우선적으로 고려하되, 만일 두 차원축소 방법 간에 분류 정확도에 차이가 적을 시, 그리고 PCA에서 주성분 추가분에 대한 정분류율 상승이 거의 없을 시에, Fisher's LDA 방법을 활용하여 차원축소를 하고, 모델을 튜닝하기로 했다. (두 차원 축소 방법의 분류적합도를 비교시에는 모형의 디폴트 옵션을 주었으며, 특성변수의 수는 최대 35개 까지만 고려하기로 했다. Fisher's LDA방법을 쓴 경우 특성변수의 수를 35 까지 올린다고 해서 정분류율이 높아지지 않는다는 것을 관찰하여, 그래프에는 특성변수의 수 16개 까지만 플롯 했으며, PCA의 경우 특성변수 16개 까지 성능이 좋지 않을 경우 특성변수 수 20개부터 플롯 하였다.)

2. 통계적 머신러닝 모델의 적용

2-1 K-nearlest Neighbors(KNN)

다음 플롯을 통해 차원 축소를 통해 얻어진 새로운 변수 수에 따라 분류 정확도에 얼마나 많이 차이가 나는지에 대해 알 수 있다. 분류 모델로서 KNN 을 사용하였다.





위의 두 플롯을 통해 알 수 있듯이, Fisher's LDA를 통해 차원 축소를 한 경우 새로운 특성변수를 추가 함으로서 얻는 분류 정확도의 추가분은 한계가 있어 보인다. (7개의 변수를 썼을 때부터 분류 정확도의 상승이 없었다.) 하지만 PCA를 통해 얻어진 주성분을 통해 KNN 모델을 적합 시킨 경우, 느리긴 하지만 분류 정확도가 확실히 상승을 하고 있다는 것을 알 수 있었다.

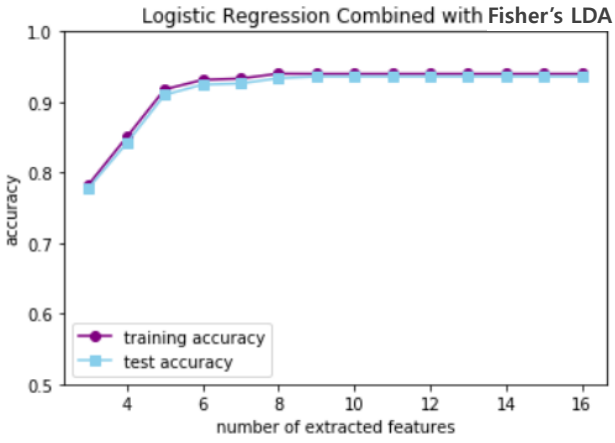
따라서 KNN의 경우에는 PCA를 통해 차원축소를 하도록 한다. 다음의 코드를 통해 cross validation 을 해본 결과 다음과 같은 최적 파라미터를 찾을 수 있었다.

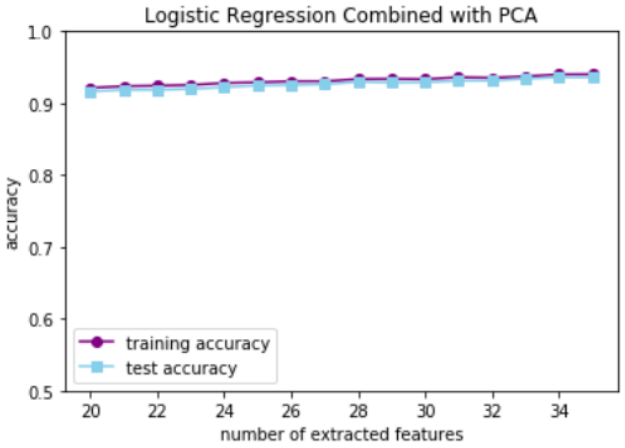
best_params	{'n_neighbors': 4, 'weights': 'distance'}

Train/test accuracy(계산시간)	1.0 / 0.94144 (2.608)

관측치에 대한 weight 를 uniform 하게 보다는 거리에 반비례 하게 주는 것이 좀더 분류 정확도를 높일 수 있었으며, 계산시간에 비해 분류정확도(94.1%)가 높은 편이었다. 과적합 문제가 다소 있는 편이었다.

2-2 Logistic Regression Classifier





Logistic Regression Classifer 의 경우에는 두 차원 축소 방법의 정 분류율의 차이가 거의 나지 않아 Fisher's LDA 를 사용하도록 하겠다(n_components=9). 주목할 점은 Logistic Regression Classifier 에서는 과적합문제가 거의 발생하지 않았다는 것이다.

Logistic Regression Classifier에서는 다양한 solver 를 사용할 수 있었으며, 각 solver마다 사용해야 하는 상황이나 사용 가능한 penalty 옵션이 모두 다르다. 따라서 각각의 solver 에 대해 cross validation을 통한 각각의 best parameter와 best score를 다음과 같이 찾아보았다.

	liblinear	Newton-cg	lbfgs	sag	saga
best_params	`{'C': 100, 'penalty': 'l1'}`	`{'C': 100, 'penalty': 'l2'}`	`{'C': 100, 'penalty': 'l2'}`	`{'C': 1, 'penalty': 'l2'}`	`{'C': 1, 'l1_ratio': 0.8, 'penalty': 'elasticnet'}`
Train/test accuracy	0.93995/ 0.93617	0.94929/ 0.94361	0.94929/ 0.94361	0.94926/ 0.94361	0.94931/ 0.9345
계산시간	1.9447	4.13202	3.2927	1.66055	15.20973

정분류율의 경우 train 셋과 test 셋의 Accuracy 93%, 94% 정도에서 solver에 관계없이 비슷한 수준이었다. (liblinear 에서는 test accuracy와 test accuracy가 다소 낮았다.) 하지만 계산시간에 있어서는 많은 차이가 났다. 가장 빨랐던 sag solver 의 경우 2초 미만 정도로 빠른 계산이 가능했으며, 이는 sag solver 가 큰 사이즈의 데이터 셋에 적합하다는 기존 지식과 동일한 결과이다. 하지만 saga solver 에서는 오히려 계산시간이 늘어났다.

(특히 lbfgs와 saga solver로 Cross validation을 할 때, max_iter =100 디폴트 값으로 학습을 시킬 경우 convergence 가 실패 했다는 에러메시지가 발생하였다.)

첨언) multi_class 옵션의 경우 liblinear solver의 경우에만 "auto"로 주었고 나머지 경우에는 "multinomial"로 주었다. 또한 penalty의 경우 liblinear와 saga 만 L1 penalty 가 가능하였고, 특히 saga solver에서만 elasticnet 패널티가 가능했다.

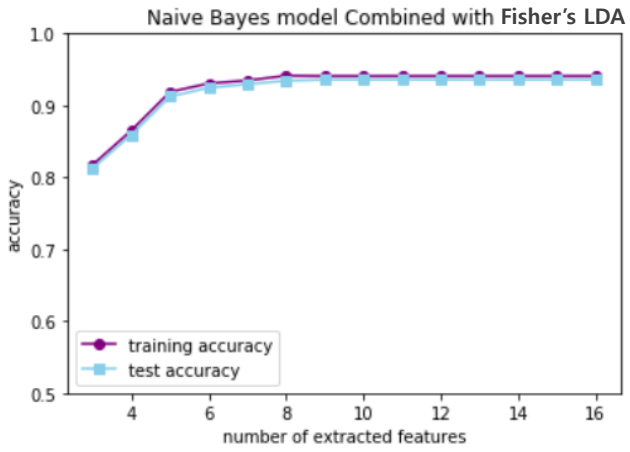
2-3 Linear Discriminant Analysis(LDA) 와 Quadratic Discriminant Analysis(QDA)

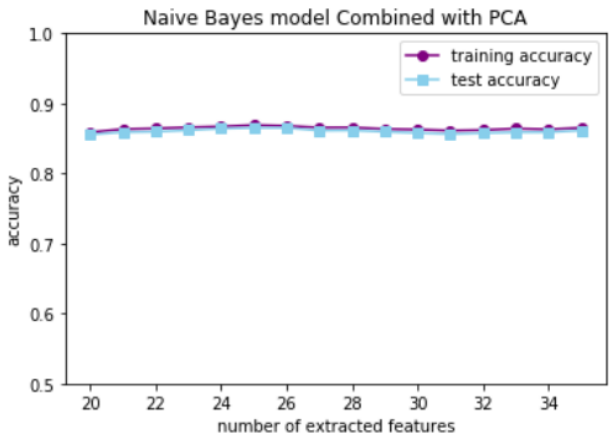
LDA와 QDA를 한번에 비교해 보도록 하겠다. 이 경우에는 따로 차원 축소 방법을 비교하지 않았다. 다음은 6차원으로 축소했을 때의 결과이다. (6개의 변수로 데이터의 90% 이상을 설명할 수 있었으며, n_component를 높게 주더라도 분류율에 의미 있는 변화가 없었다.) Linear Discriminant Analysis의 경우 "lsqr", "eigen", 그리고 "svd" solver 옵션이 있었다. "lsqr"과 "eigen" solver의 경우 shrinkage 옵션을 줄 수 있었지만("auto"로 지정했다.) "svd"의 경우에 호환되지 않는다.

	LDA(solver =lsqr)	LDA(solver =eigen)	LDA(solver =svd)	QDA
Train/test accuracy	0.92512/ 0.91844	0.92512/ 0.91844	0.92507/ 0.91844	0.94183/ 0.93578
계산시간	0.07591	0.10673	0.06081	0.08274

LDA의 경우에 세 가지 종류의 solver 모두 거의 동일한 train과 test 분류 정확도를 보였다. 하지만 계산 속도에서 약간의 차이가 난다는 사실을 알 수 있었는데, svd의 경우 계산 시간이 가장 적게 소요 되었다. 가장 높은 정분류율을 보인 것은 QDA를 통한 분류였으며, 계산속도도 두번째로 빨랐으므로, 이 데이터에서는 QDA가 좀 더 우수한 성능을 보였다고 생각한다. 과적합이 1%내로 발생함을 관찰했다.

2-4 Naïve Bayes Model





플롯에서 알 수 있듯이, 학습할 주성분의 개수를 많이 늘리더라도, LDA를 통한 차원 축소가 나이브 베이즈 모델

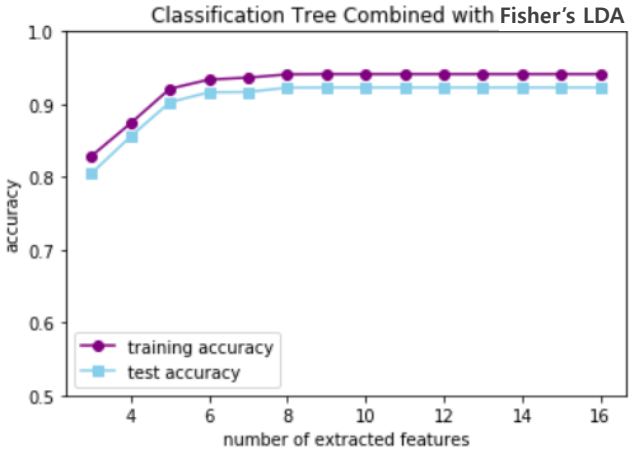
에서 성능이 월등히 좋았다는 것을 알 수 있었다. 또한 나이브 베이즈 모델에서는 과적합 문제가 적었다는 점을 주목했다.

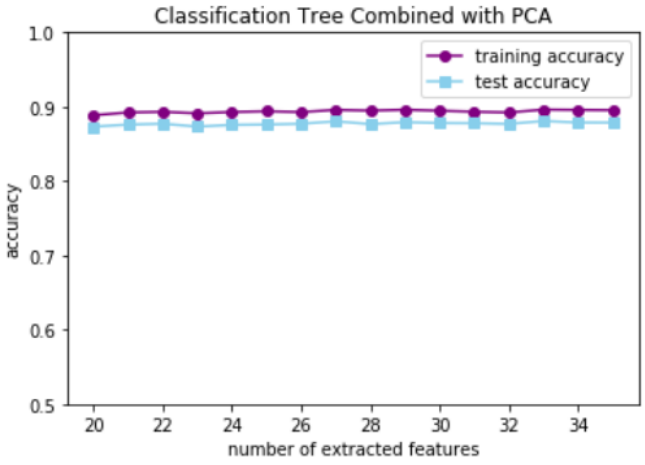
더불어 나이브 베이즈 모델의 경우, 특성변수의 차원이 커지더라도 꽤 빠른 속도로 분류가 수행되는 경험을 여럿 해본적이 있었다. 따라서 특성변수의 차원축소를 하지 않고 분류했을 경우와, 특성변수의 차원축소를 한 경우의 분류를 비교해 보았다.

	차원 축소를 하지 않은 경우	차원 축소를 한 경우
Train/test accuracy	0.72667/ 0.72011	0.94089/ 0.93577
계산시간	6.76478	0.15159

차원 축소를 하지 않은 경우, 차원 축소를 했을 때 보다 오히려 분류 정확도가 상당히 낮아지는 것을 볼 수 있었다. 모델을 적합시킬 때 중요한 것은 변수의 개수가 아닌 설명력이며, 따라서 Kannada digits 데이터와 같이 특성변수의 차원이 큰 경우 적당한 수준의 차원 축소는 필수적이란 사실을 알 수 있었다.

2-5 Classification Tree

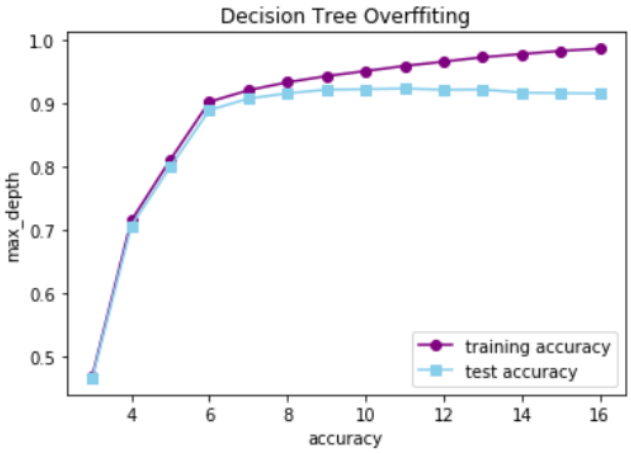




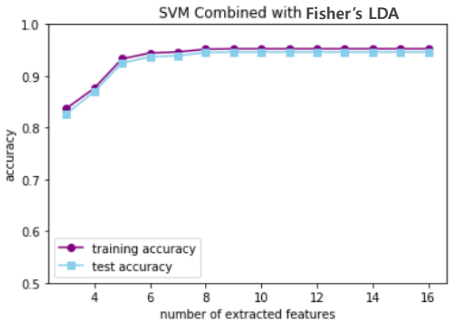
분류나무 모델의 경우에도 Fisher's LDA를 통한 차원 축소를 통해 모델을 적합시키는 편이 계산시간상으로나, 정분류율 적인 면에서 훨씬 좋았다고 할 수 있다. 따라서 Fisher's LDA를 통한 차원축소를 하기로 한다. (9차원으로 축소함)

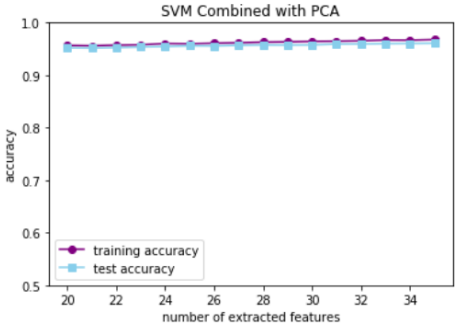
	Gini 를 기준으로	Entropy를 기준으로
best_params	`{'max_depth': 11}`	`'max_depth': 10}`
Train/test accuracy	0.9666/ 0.93122	0.96071/ 0.92972
계산시간	0.62006	1.77238

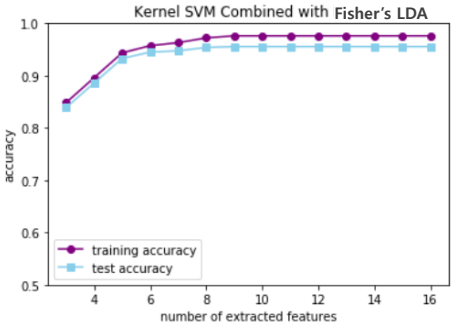
엔트로피보다는 Gini를 기준으로 모델을 적합시켰을 때 train과 test 셋의 분류 정확도가 좀 더 높았으며 계산시간도 빨랐다는 것을 알 수 있었다. 주목할 점은 분류나무 모델은 다른 모델에 비해 과적합의 정도가 좀 더 심각하다는 것이다. 이는 max_depth 옵션 값을 크게 줄수록 심각해 진다는 것을 통해서도 알 수 있었다.

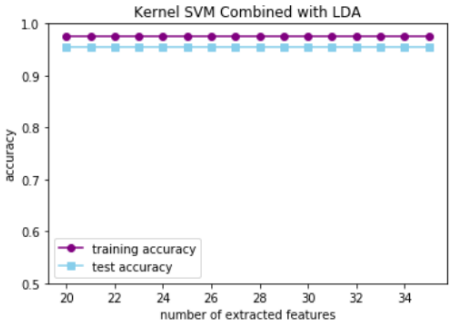


2-6 SVM & Kernel PCA





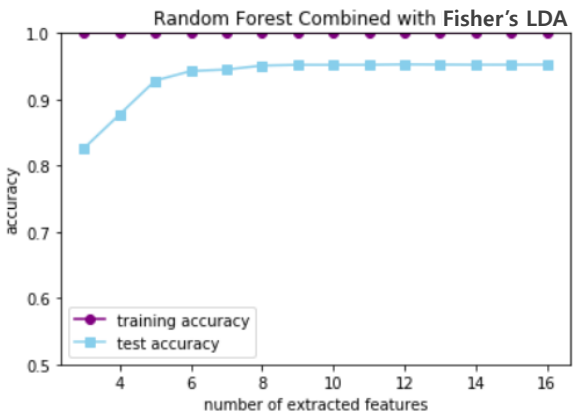


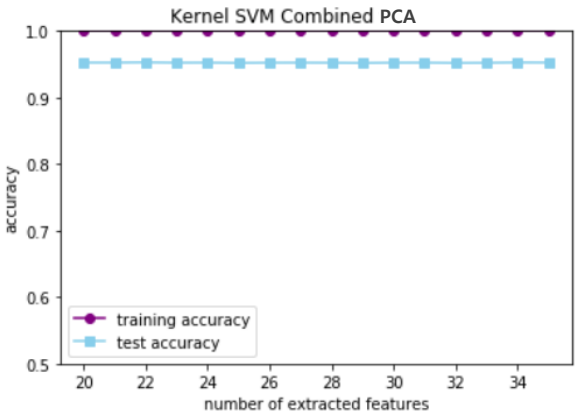


SVM의 경우 PCA를 통해 차원을 축소한 경우에 가장 정분류율이 높았으며, SVM 보다는 Kernel SVM을 사용했을 때 과적합 문제가 좀 더 심각했다. 따라서 여기서는 PCA를 통해 차원 축소를 하고 (35개 특성변수 추출) 튜닝을 하기로 했다. 계산시간이 많이 걸리는 편이었지만 분류 정확도가 높은 편이었다.

best_params	`{'C': 1, 'kernel': 'linear'}`
Train/test accuracy(계산시간)	0.96783 / 0.95978 (35.7878)

2-7 Random Forest





랜덤 포레스트 모델의 경우 두 차원 축소간의 정분류율이 거의 차이가 나지 않았다. 따라서 Fisher's LDA를 통한 차원축소를 하기로 한다(n_componets= 9). 또한 랜덤 포레스트 모형은 다른 모델에 비해 과적합의 정도가 심하다는 것을 알 수 이었다. 랜덤 포레스트의 경우 따로 튜닝을 거치지 않았으며, fit과 prediction 을 위한 계산시간은 총 약 131초가 걸렸으며, 정분류율은 train과 test 데이터 각각 1.0 과 0.95206 이었으므로, 과적합문제가 있을지라도 성능은 매우 우수했다.

2-9 Gradient Boosting

Gradient Boosting의경우 따로 차원축소 방법에 대한 비교를 하진 않았으며 fisher's LDA를 통한 차원축소(n_components=9)를 이용했다(앙상블도 마찬가지). fit에서 prediction 까지 시간은 408 초 가량 소비되었으며, 정분류율은 train과 test 셋 각각 1.0과 0.9516 이었다. 이경우에도 과적합은 다른 모델보다는 심한 편이었다.

첨언) light GBM을 실행할 수 있는 lightgbm 팩키지를 사용해 모델을 적합시킬수도 있다. 경험상으로 xgboost보다 계산속도가 빠른 경우가 많으며, loss 값을 iteration 마다 볼 수 있으므로, loss가 떨어지는 형태를 보고 파라미터 설정이 적절하지 않다고 느낄 시 미리 중지 할 수 있다는 장점이 있었다. 계산시간은 약 204초가 걸렸으며 train 과 test셋의 정분류율은 각각 1.0과 0.954 로 오히려 Gradient boosting 보다 더 나았다. 다음은 코드와 아웃풋의 예시이다.

```
import lightgbm as lgb

params = {'learning_rate': 0.1,

          'max_depth': 4 .....}

train_ds = lgb.Dataset(X_train_lda, label =y_train)

test_ds = lgb.Dataset(X_test_lda, label = y_test)

model = lgb.train(params, train_ds, 10000, test_ds,
verbose_eval=100, early_stopping_rounds=500)
```

```
[1900]  valid_0's multi_error: 0.0460556
[2000]  valid_0's multi_error: 0.0463333
[2100]  valid_0's multi_error: 0.0465
[2200]  valid_0's multi_error: 0.0463333
[2300]  valid_0's multi_error: 0.0463333
[2400]  valid_0's multi_error: 0.0465
[2500]  valid_0's multi_error: 0.0463889
[2600]  valid_0's multi_error: 0.0465
[2700]  valid_0's multi_error: 0.0462222
Early stopping, best iteration is:
[2277]  valid_0's multi_error: 0.0459444
time : 203.88825869560242
```

2-10 Ensemble Model

대부분의 모델에서 Fisher's LDA에 의한 차원축소가 효과적이었으므로 Fisher's LDA를 통해 뽑은 새로운 특성변수를 사용하기로 한다. 먼저 위 모델 중에서 성능이 좋은 편에 속했던 SVM과 Random Forest, Gradient Boosting을 앙상블 하였다. 또한 성능이 상대적으로 안 좋은 모델이라도 앙상블 했을 때 정분류율이 올라가는지 궁금하여 Classification Tree와 Naïve Bayes 모델, 그리고 QDA를 앙상블 해보았다.

2-10-1 SVM, Random Forest, Gradient Boosting

모델	아웃풋
rfc = RandomForestClassifier(n_estimators=100) svm = SVC(C=1, kernel="linear") gbc = GradientBoostingClassifier() voting_high=VotingClassifier(estimators=[('rfc', rfc), ('svc', svm), ('gbc', gbc)],voting='hard')	RandomForestClassifier 0.9516111111111111 SVC 0.9461666666666667 GradientBoostingClassifier 0.9456666666666667 VotingClassifier 0.9513333333333334

2-10-2 Classification Tree, Naïve Bayes, QDA

모델	아웃풋
tree = DecisionTreeClassifier(max_depth = 11) Gnb = GaussianNB() qda = QuadraticDiscriminantAnalysis() voting_low=VotingClassifier(estimators=[('tree', tree), ('Gnb', Gnb), ('qda', qda)],voting='hard')	DecisionTreeClassifier 0.9305555555555556 GaussianNB 0.9357777777777778 QuadraticDiscriminantAnalysis 0.9428888888888889 VotingClassifier 0.9427222222222222

예상외로 두 가지 경우 모두 가장 성능이 좋았던 모델은 앙상블이 아니었다. 2-10-1 모델에서는 랜덤 포레스트 모델이 가장 정분류율이 높았으며, 2-10-2 모델에서는 QDA 모델에서 가장 높았다. 하지만 앙상블 모델과의 차이가 매우 미미하며, 앙상블 모델이 확실히 나머지 모델보다는 성능이 좋았다는 사실을 알 수 있었다. 따라서 앙상블 모델을 사용할 때 정분류율을 항상 높이는 것은 아니지만, 어떤 모델을 써야 할지 모르는 상황에서는 확실히 좋은 전략이라고 결론 내렸다.

3. 결론

- 특성변수의 수가 너무 많다면 PCA나 Fisher's LDA로 차원 축소하는 것이 정확도와 계산시간을 개선할 수 있다. (예 : 나이브 베이즈 모델)
- 특성변수의 수가 너무 많다면 Fisher's LDA가 좀더 차원축소를 효과적으로 할 수 있는 편이지만, 분류 정확도를 고려한다면 PCA를 택해 특성변수를 많이 사용하는 것이 좋은 경우 또한 있었다. (예 : KNN). 또한 Fisher's LDA를 통해 차원 축소한 변수들의 수를 늘리는 전략은 PCA에서 보다 한계점이 명확했다.
- 분류 정확도가 가장 높은 모형은 PCA를 활용한 SVM이었고, 랜덤포레스트와 GBM 도 성능이 좋았다.
- 트리모델, 랜덤포레스트, Kernel PCA, KNN은 과적합 문제가 다른 모델보다 심한 편이었다.