CS 178 Project: Predicting Rainfall

https://inclass.kaggle.com/c/cs178-project-2016

Xinglong Bai (70713813)

Minjun Yu (83601665)

Andrew Fischer (60751837)

**Abstract**

We explored k-means clustering for adding features to decision trees, feature selection for the general k-nearest-neighbor method, and support vector regression. As the project progressed, feature adding using clustering was dropped, due to bad results. The bagged forest model proved to be useful, even on its own. As standard predictors, we also considered linear regression and k-nearest neighbors. Next, we tested support vector regressors using the radial basis function method. For our final model, we used linear regression, knn, and a bagged forest. Our RMSE scores from Kaggle hovered around 0.616.

**Introduction**

We chose to predict rainfall based on a number of features given in the Kaggle data. In addition to writing code, we used code both from class work and from the popular *scikit-learn* Python package. The project was divided into several parts. Andrew Fischer worked on clustering. Minjun Yu worked on support vector machines. Xinglong Bai worked on k-nearest neighbors. Finally, all members participated in combining learners into an ensemble.

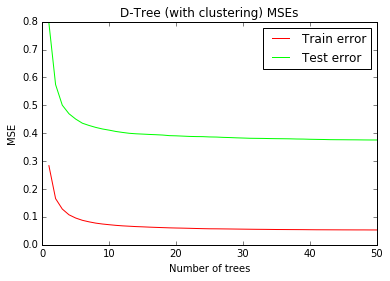
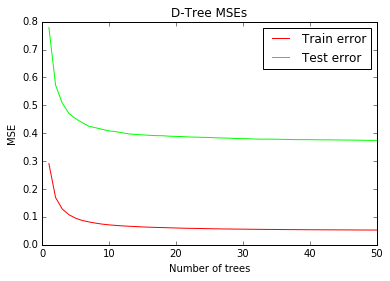
**Planned Process**

We planned to explore support vector machines, ensembles, and feature adding from clustering. First, we would use k-means clustering to find some structure within the data and add features to our existing data. Second, we would explore support vector machines, more specifically, support vector regressions. Finally, we would gather multiple learners into a single ensemble, from which we would predict the target values.

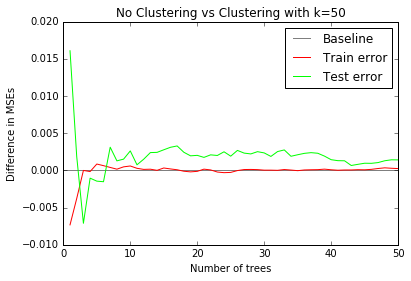
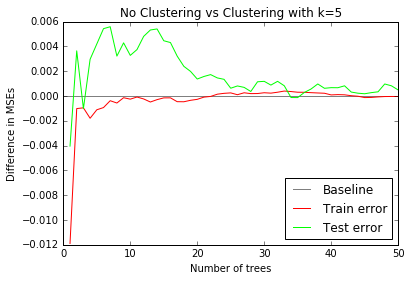
**Actual Process**

**Clustering**

K-means clustering was chosen, since it allows out-of-sample (test) data to be labelled. Random forests (using bagging) were trained both with and without the added features. After varying the numbers of clusters ranging from 5 to 50, we found that k-means clustering produced features that reduced prediction accuracy. Model accuracy dropped as the number of added features increased. For example, a forest without clustering produced an MSE of 0.37424 on the validation data. Forests with 5 and 50 clusters produced MSEs of 0.37473 and 0.37565, respectively. The graphs below show train and test error of random forests as a function of the number of trees. The first forest only trained on the given data, while the second forest trained on data with 50 added features. The graphs look similar, so the results were not drastically changed, but the MSE with clustering was consistently higher as the number of trees approached 50.



The following graphs compare random forests that use clustering features (k=5, k=50) with the forest that uses only the given data. Baseline represents the forest without clustering, so forests with values above it indicate worse performance. Based on these graphs, we chose not to use clustering for our final submission.



It should be noted that clustering produces binary features, but decision tree regressors were trained on all features in the same way. This probably explains the sub-par performance for predicting a continuous target.

**Bagged Forest**

A large bagged forest (n=500) was trained on the data. After clustering failed to produce desirable results, more time was given for training a boost-aggregated forest. To train the forest, each tree regressor was given bootstrapped data and only considered a random selection of 50 features. Its mean squared error (MSE) when using the validation split seemed to asymptotically approach 0.370, regardless of how many trees were used. A large forest of bagged trees has no overfitting problems with the rainfall data. However, a forest with less than 10 trees struggles to model the data well.

**K Nearest Neighbors (Focus on feature selection)**

The knn’s training process is very simple, just store the training data points in a data structure. When it makes a prediction, it finds the k nearest neighbors by calculating the euclidian distances from data point xi to other data points Sigma((x1-xi)\*\*2 + (x2 - xi)\*\*2 + ...). Thus most of the running time is spent on finding the k nearest neighbors. The value of k and the number of features can decide the running time and performance. There are three steps to optimize the knn predictor.

Step 1: Finding the best k value

First train the knn learner with 75 percent of the training dataset, then calculate the validation mses with different k values

k = 1 VMSE = 0.920681498282  
k = 5 VMSE = 0.57244645001  
k = 25 VMSE = 0.503324974713  
k = 40 VMSE = 0.500598024389  
k = 75 VMSE = 0.49881196938  
k = 80 VMSE = 0.498803577424  
k = 85 VMSE = 0.499106960447  
k = 90 VMSE = 0.49900214539  
k = 100 VMSE = 0.499833659587  
k = 120 VMSE = 0.500476188954

When k is 1, the model is too complex and overfits; as the k becomes very large, the knn model experiences underfitting. A k value of 80 can make the predictor get the lowest validation mean squared error.

Step 2: Feature selection

There are 91 features of each observation. There are some features that are not important, and some features are important. When training the model only with the features which are important, we expect the prediction to perform better. First train a knn model with a single feature for each of the features, and output the validation mses. Then find the feature with the best performance. Sort the features according to the VMSE performance of each feature. The feature with the lowest vmse will be the top of the feature list, which is the most important.

##------------------------feature selection

knnpredict = []

knnVMse = []

xs = [i for i in range(n)] # index of features 1-91

for i in range(n):

knn = KNeighborsRegressor(n\_neighbors = bestK)

knn.fit(Xtr[:, i].reshape(Xtr.shape[0], 1), Ytr)

p = knn.predict(Xte[:, i].reshape(Xte.shape[0], 1)) #make prediction with one feature

knnpredict.append(p)

knnVMse.append(mean\_squared\_error(Yte, p)) # count the Validation mse of that feature

zipped = zip(xs, knnVMse) #zip the features’ index and their associated VMSE

featureRank = sorted(zipped, key = lambda x: x[1]) #sort the features according to VMSE

##---------------------------------

The best 10 features and the VMSE [(ith feature, VMSE ), …..]:

[(2, 0.52485), (23, 0.52485), (3, 0.5260), (5, 0.5317), (27, 0.5387), (9, 0.54445), (8, 0.5484), (19, 0.5507), (10, 0.5712), (15, 0.5831)]

We did not use all of the 91 features. After doing a feature ranking, the knn model was trained with the first n features which performed best during the feature ranking. We tried to use different numbers of features to train knn and observe the performance.

Validation mse:

""" ranked feature not ranked ranked feature not ranked """

"""60 features: 0.47711, 0.4772 50 features: 0.45138, 0.4515

40 features: 0.4556 0.453370 35 features: 0.45016, 0.455

30 features: 0.45016, 0.455 20 features: 0.45492, 0.4570

15 features: 0.46504 2 features: 0.52485

"""

We found that when using 30 features, the validation mse is the lowest. After we did the feature performance ranking and chose the 30 features with the best performance, the result performed better than using the 30 features without feature ranking.

The features we chose are:

selectedFeature = (2, 23, 3, 5, 27, 9, 8, 19, 10, 15, 7, 55, 83, 6, 84, 4, 0, 67, 63, 11, 87, 88, 35, 71, 39, 43, 40, 59, 42, 36)

Train knn model:

knn = KNeighborsRegressor(n\_neighbors = 80)

knn.fit(X[:, selectedFeature], Y)

Makeprediction:

predict = knn.predict(Xte[:, selectedFeature])

output: Validation mse is 0.435361536177 expect meets

Step 3 Boosting

In order to improve the performance of knn a little more, gradient boosting was used to integrate multiple knn learners. For each knn learner, we trained the training data with only one feature, selected from the best 30 features in the feature ranking.

for i in range(30):

knn = KNeighborsRegressor(n\_neighbors = bestK) # best k value is 80

features = [featureRank[i+j][0] for j in range(1)]

selected = tuple(features)

knn.fit(Xtr[:, selected], dY)

Ptr[:, i] = knn.predict(Xtr[:, selected])

Ptv[:, i] = knn.predict(Xte[:, selected])

dY -= Ptr[:, i]

prediction = Ptv[:, 0:30].sum(axis = 1) + mu

However, the Validation mse is 0.555311257446, which is higher than expected. When we made predictions with only 15 learners which were trained in the same way, the error dropped to 0.525202663331. Thus, boosting the knn doesn’t meet expectations, and we did not use boosting on knn.

**Support Vector Machine**

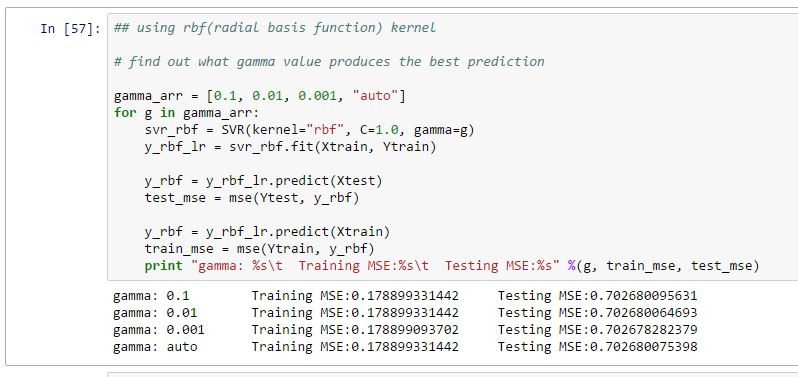
We tried Support Vector Machines, more specifically, using SVR (Support Vector Regression). There are mainly three kinds of kernels that we tried: 1. Linear kernel, 2. Polynomial kernel, and 3. RBF (Radial Basis Function) kernel. Both Linear kernel and Polynomial kernel run very slowly with the Kaggle data and the performance is not particularly promising, so we decided to focus on RBF.

SVR has 2 major parameters which are Gamma and C value.

**SVM - Gamma**

Gamma defines how far the influence of a single data point reaches. The smaller the gamma value is, the farther the data point reaches.

For gamma value, we tried 0.1, 0.01, 0.001 and “auto” (1 / number of data points). The result is as follows:

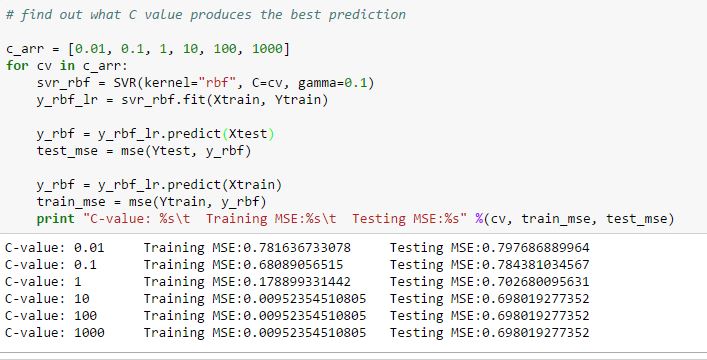


As the illustration suggests, the gamma value does not affect the performance of our prediction greatly, so we picked gamma value 0.1.

**SVM - C**

The C parameter trades off misclassification of training examples against simplicity of the decision surface.The greater the C value is, the more complex the model will be.

For C value, we tried 0.01, 0.1, 1, 10, 100 and 1000. The result is as follows:



As the illustration suggests, the lowest testing MSE occurs when C is of value 1 or above. And when C value is 1 or under, for instance, 0.01, the model is underfitting. To avoid underfitting or overfitting, C should be of value 1.

In general, the SVR model performs comparatively well when C is 1 and gamma is 0.1.

**Univariate feature selection**

As shown in the previous section, the MSE value is still high compared to other learners such as knn. Given that the data has 91 features which is a great amount, there must be some features that are not relevant to the prediction. This is where feature selection comes into play.

Univariate feature selection works by selecting the best features based on univariate statistical tests. It can be seen as a preprocessing step to a learner.

To do this, we pick k = 1 … 91 best features and find out which k value yields the best result. The result is as follows:

1 best features selected tr\_mse:0.452121610627 te\_mse:0.63258586207

2 best features selected tr\_mse:0.432153910622 te\_mse:0.646465545602

3 best features selected tr\_mse:0.181777726663 te\_mse:0.69112547152

4 best features selected tr\_mse:0.178699644023 te\_mse:0.701219463479

5 best features selected tr\_mse:0.17900585768 te\_mse:0.702647309031

6 best features selected tr\_mse:0.179010859704 te\_mse:0.702671975863

7 best features selected tr\_mse:0.179010818141 te\_mse:0.702672221123

8 best features selected tr\_mse:0.179010818141 te\_mse:0.70267232156

9 best features selected tr\_mse:0.179010818141 te\_mse:0.70267232156

10 best features selected tr\_mse:0.179010817866 te\_mse:0.702672390749

11 best features selected tr\_mse:0.178923020922 te\_mse:0.702659622003

…

…

Surprisingly, the prediction performs best when only one best feature is selected. It clearly does not follow normal expectations because it does not conform to knn or decision tree predictors we made.

SVM might not be an ideal way to predict this kaggle data.

**Final Results**

Individual training produced the following results:

1 Knn Validation MSE = 0.435361536177

2 Linear Regression MSE = 0.424905288513

3 SVM Regression MSE = 0.724217765211

4 Bagged Forest (500) MSE = 0.370257778669

The final ensemble of learners produced the following results:

1 Validation MSE = 0.189044049728

2 Validation MSE without knn = 0.187280227997

3 Validation MSE(Bagged forest(50 tree), knn, svr) = 0.297965592958

4 Kaggle score: **0.619200 (Bagged Forest, linear regression, knn) 0.61548 (Bagged Forest, knn) 0.65466 (Bagged Forest, knn, svr)**