A1-Iyer-Chen

February 24, 2017

1 Foundations of Data Mining: Assignment 1

Suraj Iyer (0866094) Simin Chen (0842556)

```
In [1]: %matplotlib inline
        from preamble import *
        plt.rcParams['savefig.dpi'] = 100
        InteractiveShell.ast_node_interactivity = "all"
```

1.1 Handwritten digit recognition (5 points, 1+2+2)

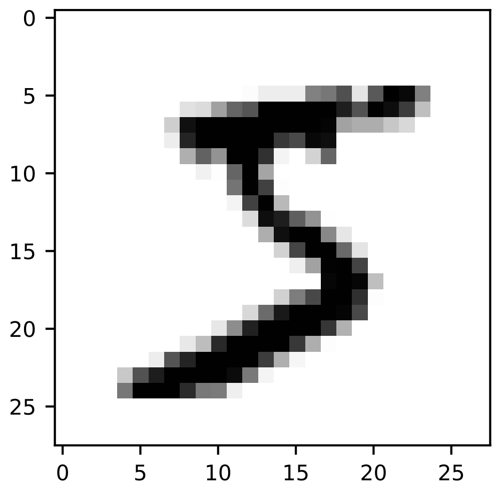
The MNIST dataset contains 70,000 images of handwritten digits (0-9) represented by 28 by 28 pixel values. We can easily download it from OpenML and visualize one of the examples:

```
In [2]: # This is a temporary read-only OpenML key. Replace with your own key later.
        oml.config.apikey = '11e82c8d91c5abece86f424369c71590'
```

```
In [3]: mnist_data = oml.datasets.get_dataset(554)  # Download MNIST data
        X, y = mnist_data.get_data(target=mnist_data.default_target_attribute)  # Get the predic
        # X, y = X[:2000], y[:2000]  # TODO: REMOVE THIS LATER
        plt.imshow(X[0].reshape(28, 28), cmap=plt.cm.gray_r)  # Take the first example, reshape
        print("Class label:", y[0])  # Print the correct class label
```

```
Out[3]: <matplotlib.image.AxesImage at 0x23bacb44a58>
```

```
Class label: 5
```



- Evaluate a k-Nearest Neighbor classifier with its default settings.
  - Use the first 60,000 examples as the training set and the last 10,000 as the test set
  - What is the predictive accuracy?
  - Find a few misclassifications, and plot them together with the true labels (as above). Are these images really hard to classify?
- Optimize the value for the number of neighbors k (keep $k < 50$) on a stratified subsample (e.g. 10%) of the data
  - Use 10-fold crossvalidation and plot k against the misclassification rate. Which value of k should you pick?
  - Do the same but with 100 bootstrapping repeats. Are the results different? Explain.
- Compare kNN against the linear classification models that we have covered in the course (logistic regression and linear SVMs).
  - First use the default hyperparameter settings.
  - Next, optimize for the degree of regularization (C) and choice of penalty (L1/L2). Again, plot the accuracy while increasing the degree of regularization for different penalties. Interpret the results.
  - Report is the optimal performance. Can you get better results than kNN?

Report all results clearly and interpret the results.
Note: while prototyping/bugfixing, you can speed up experiments by taking a smaller sample of the data, but report your results as indicated above.

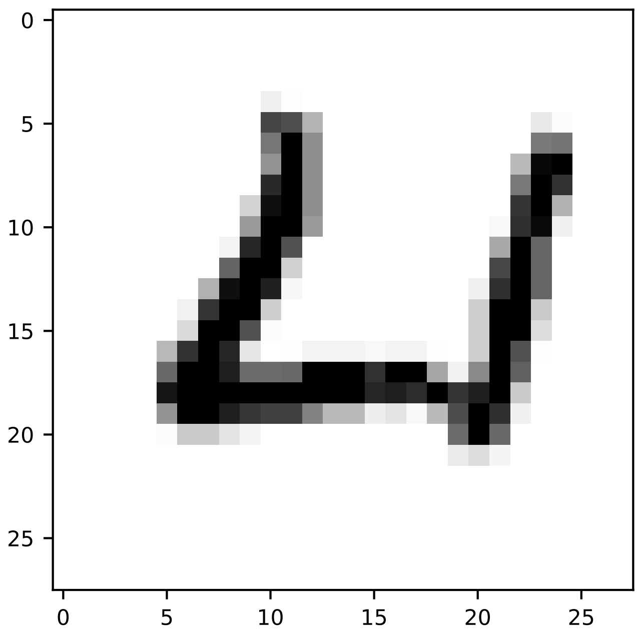
Evaluate a k-Nearest Neighbor classifier with its default settings

```
In [60]: from sklearn.neighbors import KNeighborsClassifier
         from sklearn.preprocessing import MinMaxScaler
```

```
min_max_scaler = MinMaxScaler()
X_train, X_test = np.split(X, [60000])
y_train, y_test = np.split(y, [60000])
# X_train, X_test = min_max_scaler.fit_transform(X_train), min_max_scaler.fit_transform
print('X_train:', X_train.shape, '| y_train:', y_train.shape, '| X_test:', X_test.shape

# Predictive accuracy
knn = KNeighborsClassifier(n_jobs=-1)
print('KNN score (predictive accuracy): %f' % knn.fit(X_train, y_train).score(X_test, y

# Finding a few misclassifications
X_misc = [(X_test[i], y, y_test[i]) for i, y in enumerate(knn.predict(X_test)) if y !=
for x, y_pred, y_true in X_misc[:5]:
    plt.imshow(x.reshape(28, 28), cmap=plt.cm.gray_r)
    plt.show()
    print("Predicted class label:", y_pred, ", True class label:", y_true)
```

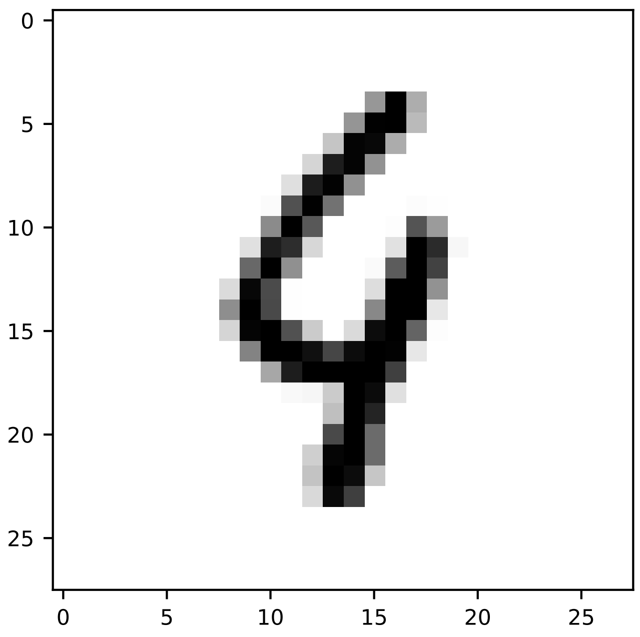
```
X_train: (60000, 784) | y_train: (60000,) | X_test: (10000, 784) | y_test: (10000,)
KNN score (predictive accuracy): 0.968800
```

```
Out[60]: <matplotlib.image.AxesImage at 0x2852cb02b38>
```



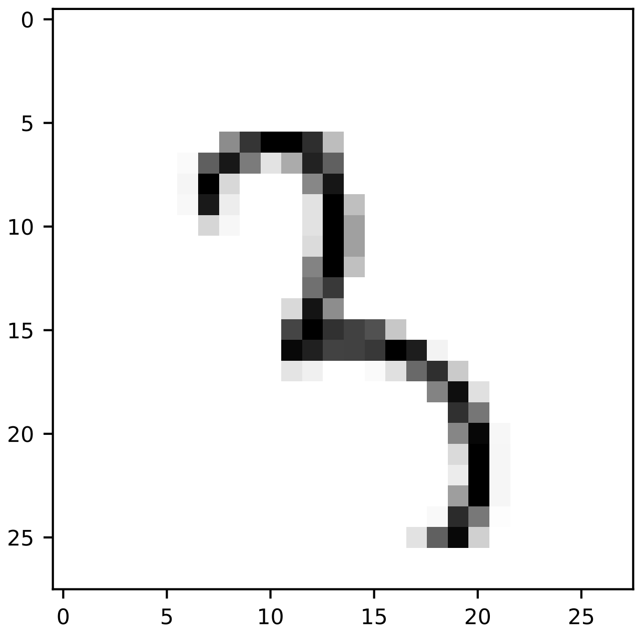
```
Predicted class label: 0 , True class label: 4
```

```
Out[60]: <matplotlib.image.AxesImage at 0x2852c44b208>
```



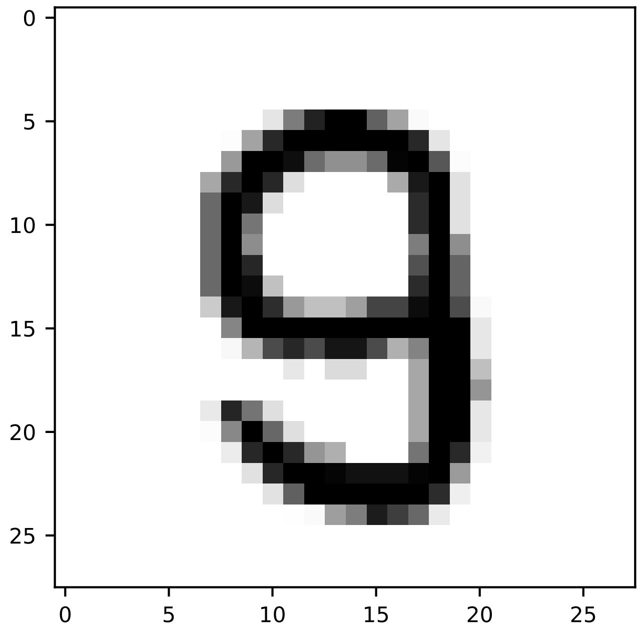
```
Predicted class label: 9 , True class label: 4
```

```
Out[60]: <matplotlib.image.AxesImage at 0x2852c841908>
```



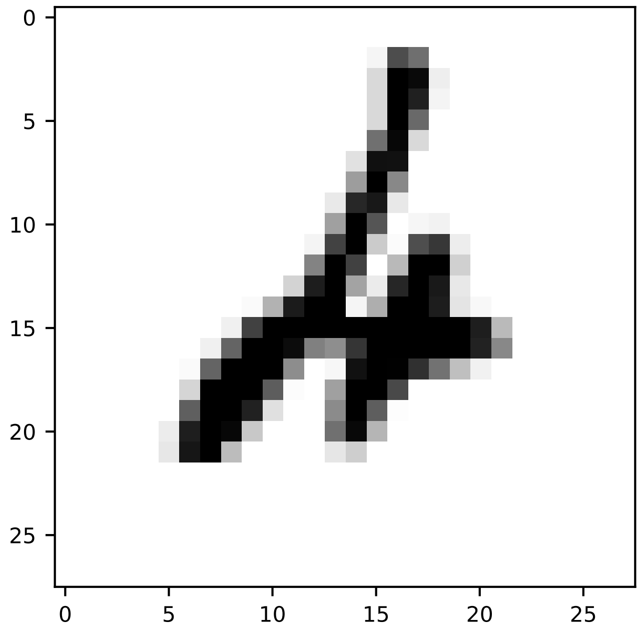
```
Predicted class label: 1 , True class label: 3
```

```
Out[60]: <matplotlib.image.AxesImage at 0x2852c8a74a8>
```



```
Predicted class label: 8 , True class label: 9
```

```
Out[60]: <matplotlib.image.AxesImage at 0x2852cb4c390>
```



```
Predicted class label: 6 , True class label: 4
```

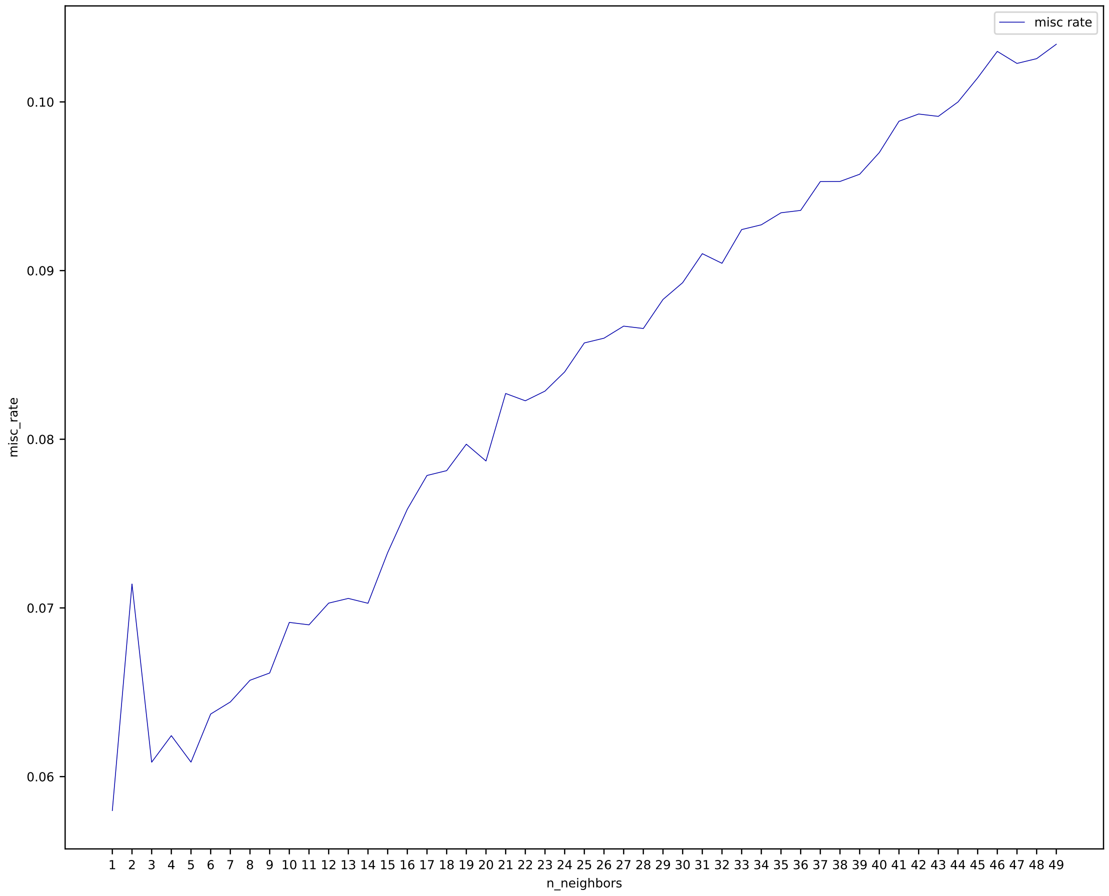
Some of the images are indeed quite difficult to recognize even by human. For example, the first misclassified 4 is not recognizable as any number, even as its predicted value. The second misclassified 4 looks quite like 9.

Optimize the value for the number of neighbors k (keep $k < 50$) on a stratified subsample (e.g. 10%) of the data

```
In [ ]: from sklearn.neighbors import KNeighborsClassifier
        from sklearn.model_selection import train_test_split, cross_val_score

        # Build a list of the training and test scores for increasing k
        misc_rate = []
        k = range(1, 50)

        # Get 10% of the data
        X_train, _, y_train, _ = train_test_split(X, y, stratify=y, train_size=.1, random_state=

        for n_neighbors in k:
            # build the model
            clf = KNeighborsClassifier(n_neighbors=n_neighbors)
            scores = cross_val_score(clf, X_train, y_train, cv=10, n_jobs=-1)
            misc_rate.append(1.-np.mean(scores))
```

```
# plot the data
plt.rcParams['figure.figsize'] = (12., 10.)
plt.plot(k, misc_rate, label="misc rate")
_ = plt.xticks(k)
_ = plt.ylabel("misc_rate")
_ = plt.xlabel("n_neighbors")
_ = plt.legend()
plt.rcParams['figure.figsize'] = (6., 4.)
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x1e392689a58>]
```



We should take $k = 1$ because it has the least misclassification rate.

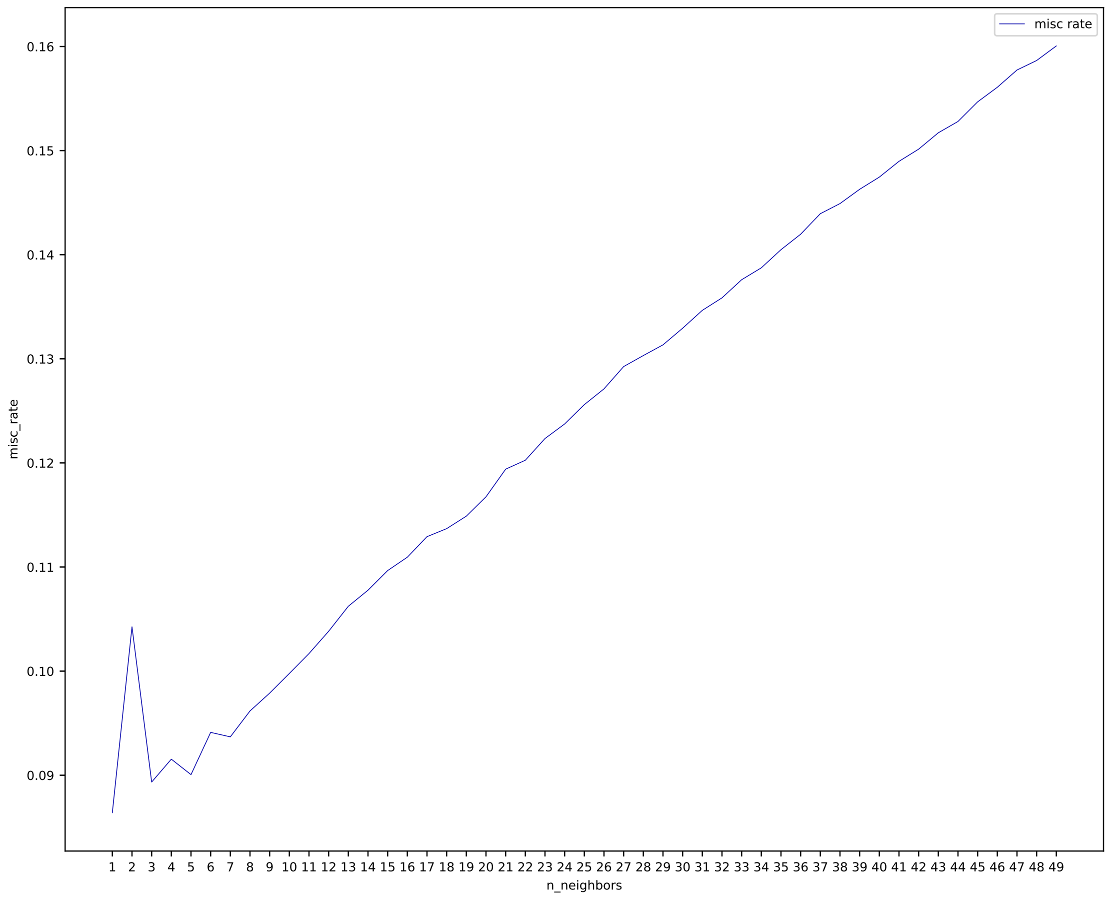
Doing the same as above but with 100 bootstrapping repeats.

```
In [27]: from sklearn.neighbors import KNeighborsClassifier
         from sklearn.model_selection import train_test_split, cross_val_score, StratifiedShuffl
         sss = StratifiedShuffleSplit(n_splits=100, train_size=0.66, random_state=66)

         # Build a list of the training and test scores for increasing k
```

```
misc_rate = []
k = range(1, 50)

# Get 10% of the data
X_train, _, y_train, _ = train_test_split(X, y, stratify=y, train_size=.05, random_stat

for i, n_neighbors in enumerate(k):
    # build the model
    print(i, end=" ")
    clf = KNeighborsClassifier(n_neighbors=n_neighbors)
    scores = cross_val_score(clf, X_train, y_train, cv=sss, n_jobs=-1)
    misc_rate.append(1-np.mean(scores))

# plot the data
plt.rcParams['figure.figsize'] = (12., 10.)
plt.plot(k, misc_rate, label="misc rate")
_ = plt.xticks(k)
_ = plt.ylabel("misc_rate")
_ = plt.xlabel("n_neighbors")
_ = plt.legend()
plt.rcParams['figure.figsize'] = (6., 4.)
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 3
```

```
Out[27]: [<matplotlib.lines.Line2D at 0x24cb31e6e10>]
```



There is more variation in the result without bootstrapping repeat, while the misclassification rate grows more smoothly as the number of neighbors increases with bootstrapping repeat. This is because there is no guarantee for the randomness of the training data. By taking 100 times bootstrapping, the cross-validation score is averaged across multiple random permutation of the training data. Therefore the result is more stable.

Compare kNN against the linear classification models that we have covered in the course (logistic regression and linear SVMs) using the default hyperparameter settings.

```
In [10]: from sklearn.neighbors import KNeighborsClassifier
         from sklearn.linear_model import LogisticRegression
         from sklearn.svm import LinearSVC
         from sklearn.model_selection import train_test_split, cross_val_score

         # Get 10% of the data
         X_train, _, y_train, _ = train_test_split(X, y, stratify=y, train_size=.1, random_state

         # Build the models
         knn, logistic, svc = KNeighborsClassifier(), LogisticRegression(), LinearSVC()
         for clf in [knn, logistic, svc]:
             scores = cross_val_score(clf, X_train, y_train, cv=10, n_jobs=-1)
             print('%s Mean cross-validation score: %f' % (clf.__class__.__name__, np.mean(score
```

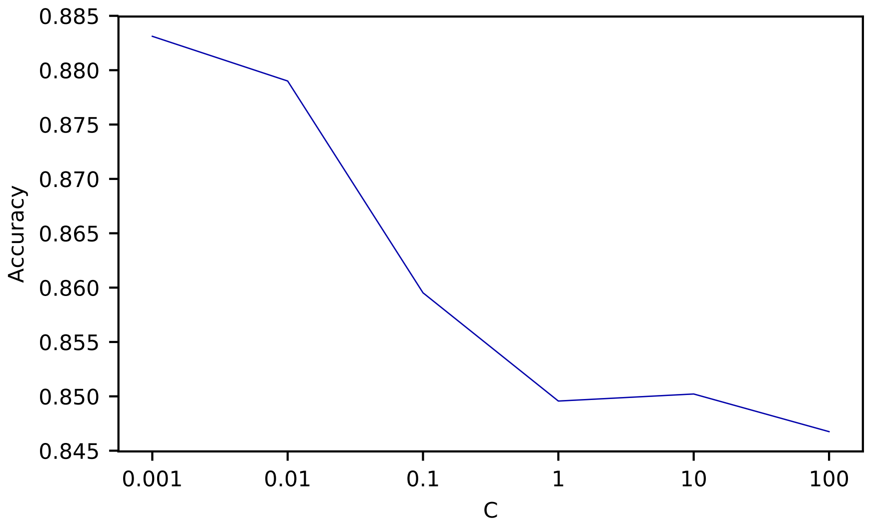
```
KNeighborsClassifier Mean cross-validation score: 0.939142
LogisticRegression Mean cross-validation score: 0.827266
LinearSVC Mean cross-validation score: 0.834135
```

Optimizing for the degree of regularization (*C*) and choice of penalty (L1/L2).

```
In [11]: from sklearn.linear_model import LogisticRegression
         from sklearn.svm import LinearSVC
         from sklearn.model_selection import train_test_split, GridSearchCV

         X_train, _, y_train, _ = train_test_split(X, y, stratify=y, train_size=.1, random_state
         X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, stratify=y_train,

         # Build the models
         logistic_l1, logistic_l2, svc = LogisticRegression(penalty='l1'), LogisticRegression(pe
         param_grid = { 'C': [0.001, 0.01, 0.1, 1, 10, 100] }

         for i, clf in enumerate([logistic_l1, logistic_l2, svc]):
             print(i, end=" ")
             grid_search = GridSearchCV(clf, param_grid, cv=5, n_jobs=-1)
             _ = grid_search.fit(X_train, y_train)
             score = grid_search.score(X_test, y_test)
             print('%s score: %f' % (clf.__class__.__name__, score))
             _ = plt.xlabel('C')
             _ = plt.xticks(np.arange(len(param_grid['C'])), param_grid['C'])
             _ = plt.ylabel('Accuracy')
             plt.plot(np.arange(len(param_grid['C'])), grid_search.cv_results_['mean_test_score'
             plt.show()
```

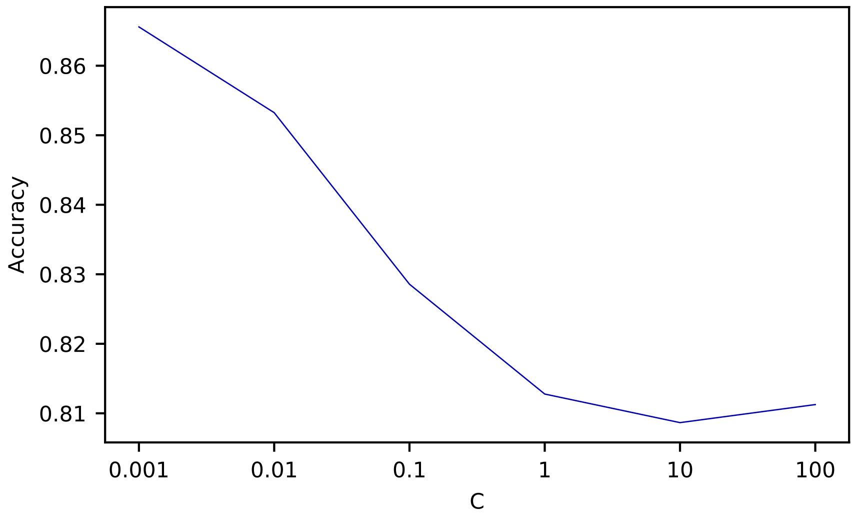
```
0 LogisticRegression score: 0.873109
```

```
Out[11]: [<matplotlib.lines.Line2D at 0x23badb9fc18>]
```



```
1 LogisticRegression score: 0.850420
```

```
Out[11]: [<matplotlib.lines.Line2D at 0x23badccf048>]
```

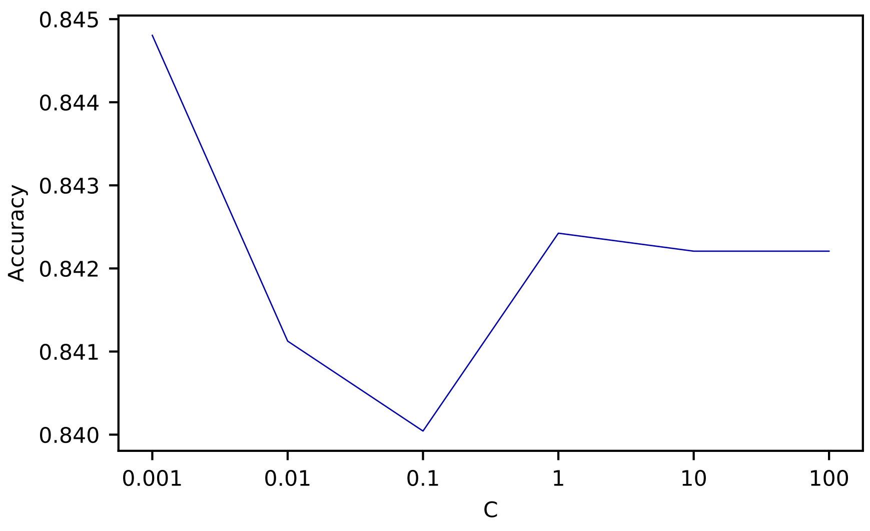


```
2
```

```
In [12]: from sklearn.svm import LinearSVC
         from sklearn.model_selection import train_test_split

         X_train, _, y_train, _ = train_test_split(X, y, stratify=y, train_size=.1, random_state
         X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, stratify=y_train,

         # Build the model
         svc = LinearSVC()
         param_grid = { 'C': [0.001, 0.01, 0.1, 1, 10, 100] }
         grid_search = GridSearchCV(svc, param_grid, cv=5, n_jobs=-1)
         _ = grid_search.fit(X_train, y_train)
         score = grid_search.score(X_test, y_test)
         print('%s score: %f' % (clf.__class__.__name__, score))
         _ = plt.xlabel('C')
         _ = plt.xticks(np.arange(len(param_grid['C'])), param_grid['C'])
         _ = plt.ylabel('Accuracy')
         plt.plot(np.arange(len(param_grid['C'])), grid_search.cv_results_['mean_test_score'])
         plt.show()
```

```
LinearSVC score: 0.830252
```

```
Out[12]: [<matplotlib.lines.Line2D at 0x23badc43ba8>]
```



From the graphs, the decreasing trend of the accuracy with increasing 'C' (lower regularization) value is clearly evident for logistic regression with both L1 and L2 penalty. The curves are very similar and shows that the choice of penalty does not really have any effect on this dataset. This trend does not hold the same way for the linear SVM model where it decreases first having global minimum for C=0.1 and then increases again till C=1 and then decreases again. This shows it has more sensitivty towards regularization than the logistic regression models for this dataset.

Based on the accuracies we have seen for the linear models, the KNN classifier performs better both using default values as well as tuned values.

1.2 Model selection (4 points (2+2))

Study how RandomForest hyperparameters interact on the Ionosphere dataset (OpenML ID 59).

- Optimize a RandomForest, varying both *n_estimators* and *max_features* at the same time. Use a nested cross-validation and a grid search (or random search) over the possible values, and measure the AUC. Explore how fine-grained this grid/random search can be, given your computational resources. What is the optimal AUC performance you find?
- Again, vary both hyperparameters, but this time use a grid search and visualize the results as a plot (heatmap) $n\_estimators \times max\_features \rightarrow AUC$ with AUC visualized as the color of the data point. Try to make the grid as fine as possible. Interpret the results. Can you explain your observations? What did you learn about tuning RandomForests?

Hint: Running this experiment can take a while, so start early and use a feasible grid/random search. Start with a coarse grid or few random search iterations. Hint: Use a log scale (1,2,4,8,16,...) for *n_estimators*. Vary *max_features* linearly between 1 and the total number of features. Note that, if you give *max_features* a float value, it will use it as the percentage of the total number of features.

```
In [39]: from sklearn.ensemble import RandomForestClassifier

         ionosphere = oml.datasets.get_dataset(59)  # Download Ionosphere data
         X, y = ionosphere.get_data(target=ionosphere.default_target_attribute)  # Get the predi
         X.shape, y.shape

Out[39]: ((351, 34), (351,))
```

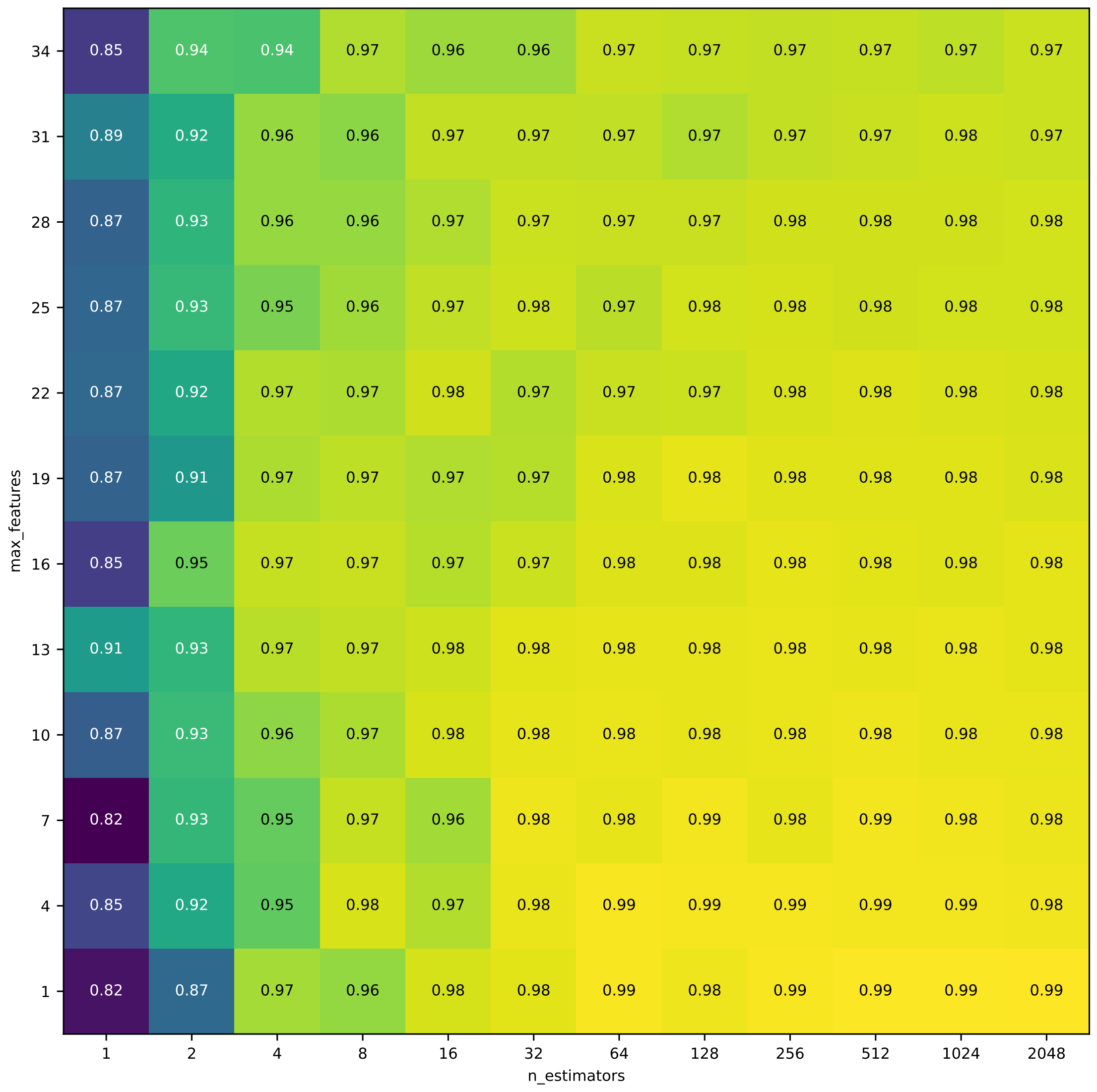
Optimize a RandomForest, varying both *n_estimators* and *max_features* at the same time using a nested cross-validation and a randomized search.

```
In [8]: from sklearn.model_selection import cross_val_score, RandomizedSearchCV

        clf = RandomForestClassifier()
        param_grid = {
            'n_estimators': [2**i for i in range(0,12)],
            'max_features': np.arange(1, X.shape[1]+1, 3)
        }
        grid_search = RandomizedSearchCV(clf, param_grid, cv=5, scoring='roc_auc', n_iter=80, n_
        scores = cross_val_score(grid_search, X, y, cv=5, scoring='roc_auc', n_jobs=-1)
        print('Scores:', scores)
        print('Mean cross-validation score:', scores.mean())

Scores: [ 0.98   0.969  0.933  1.     0.992]
Mean cross-validation score: 0.975001709402
```

When $n\_iter = 20$, we are able to get the best results within acceptable time limit. We tested with higher iterations but they gave negligable improvements to the performance but at the same time took way longer to train. So, we decided to stick with 20 iterations. The optimal AUC score we found was 1.

Again, vary both hyperparameters, but this time use a grid search and visualize the results as a plot (heatmap)

```
In [40]: from sklearn.model_selection import train_test_split, GridSearchCV

         X_train, _, y_train, _ = train_test_split(X, y, stratify=y, train_size=.66, random_stat
         clf = RandomForestClassifier()
         param_grid = {
             'n_estimators': [2**i for i in range(0, 12)],
             'max_features': np.arange(1, X.shape[1]+1, 3)
         }
         grid_search = GridSearchCV(clf, param_grid, cv=5, scoring='roc_auc', n_jobs=-1)
         _ = grid_search.fit(X_train, y_train)
         results = grid_search.cv_results_
         scores = np.array(results['mean_test_score']).reshape(len(param_grid['max_features']),

         # Display the heatmap
         plt.rcParams['figure.figsize'] = (12., 10.)
         mglearn.tools.heatmap(scores, xlabel='n_estimators', xticklabels=param_grid['n_estimato
                               ylabel='max_features', yticklabels=param_grid['max_features'], cm
         plt.rcParams['figure.figsize'] = (6., 4.)

Out[40]: <matplotlib.collections.PolyCollection at 0x1e3e20210b8>
```



max_features \ n_estimators	1	2	4	8	16	32	64	128	256	512	1024	2048
34	0.85	0.94	0.94	0.97	0.96	0.96	0.97	0.97	0.97	0.97	0.97	0.97
31	0.89	0.92	0.96	0.96	0.97	0.97	0.97	0.97	0.97	0.97	0.98	0.97
28	0.87	0.93	0.96	0.96	0.97	0.97	0.97	0.97	0.98	0.98	0.98	0.98
25	0.87	0.93	0.95	0.96	0.97	0.98	0.97	0.98	0.98	0.98	0.98	0.98
22	0.87	0.92	0.97	0.97	0.98	0.97	0.97	0.97	0.98	0.98	0.98	0.98
19	0.87	0.91	0.97	0.97	0.97	0.97	0.98	0.98	0.98	0.98	0.98	0.98
16	0.85	0.95	0.97	0.97	0.97	0.97	0.98	0.98	0.98	0.98	0.98	0.98
13	0.91	0.93	0.97	0.97	0.98	0.98	0.98	0.98	0.98	0.98	0.98	0.98
10	0.87	0.93	0.96	0.97	0.98	0.98	0.98	0.98	0.98	0.98	0.98	0.98
7	0.82	0.93	0.95	0.97	0.96	0.98	0.98	0.99	0.98	0.99	0.98	0.98
4	0.85	0.92	0.95	0.98	0.97	0.98	0.99	0.99	0.99	0.99	0.99	0.98
1	0.82	0.87	0.97	0.96	0.98	0.98	0.99	0.98	0.99	0.99	0.99	0.99

The heat map shows that for this specific data set, the more trees in the RandomForest model the better. With $n\_estimators \geq 16$, the score does not change much anymore. When the number of trees is fixed, the less $max\_features$, the better for when $n\_estimators \geq 16$ while for $n\_estimators \leq 8$, more optimal results come when $max\_features$ is somewhere in between. Normally it would be expected that with more number of features to choose from, we should get better results but obviously based on the results, this is not the case. Apparently, by introducing more number of features, the number of differences between trees generated reduces, thereby decreasing diversity of the forest. The lesser the diversity, the more chances of incorrect predictions if the model is wrong. On the other hand, when there are not enough trees, the number of features plays relatively greater role.

1.3 Decision tree heuristics (1 point)

Consider the toy training set created below. It predicts whether your date agrees to go out with you depending on the weather.

Learn a decision tree:

- Implement functions to calculate entropy and information gain
- What is the class entropy for the entire dataset? What is the information gain when you split the data using the *Water* feature?
- Implement a basic decision tree:
  - Select a feature to split on according to its information gain. If multiple features are equally good, select the leftmost one.
  - Split the data and repeat until the tree is complete.
  - Print out the results (nodes and splits).
- Now train a scikit-learn decision tree on the same data. Do you get the same result? Explain.

```
In [38]: import pandas as pd
         import numpy as np

         df = pd.DataFrame({"Sky": ['sunny', 'sunny', 'rainy', 'sunny', 'sunny'],
                            "AirTemp": ['warm', 'warm', 'warm', 'cold', 'warm'],
                            "Humidity": ['normal', 'high', 'high', 'high', 'normal'],
                            "Wind": ['strong', 'strong', 'strong', 'strong', 'weak'],
                            "Water": ['warm', 'warm', 'cool', 'warm', 'warm'],
                            "Forecast": ['same', 'same', 'change', 'change', 'same'],
                            "Date?": ['yes', 'yes', 'no', 'yes', 'no']
                            })
         df = df[['Sky', 'AirTemp', 'Humidity', 'Wind', 'Water', 'Forecast', 'Date?']]  # Fix co
         df  # print

         # One-hot encode
         cols_to_transform = ['Sky', 'AirTemp', 'Humidity', 'Wind', 'Water', 'Forecast', 'Date?'
         df1 = pd.get_dummies(df, columns=cols_to_transform)
         df1.drop(['Sky_rainy', 'AirTemp_cold', 'Humidity_high', 'Wind_strong',
                   'Water_cool', 'Forecast_change', 'Date?_no'], axis=1, inplace=True)
         df1
```

```
Out[38]:      Sky AirTemp Humidity    Wind Water Forecast Date?
         0  sunny    warm   normal  strong  warm     same   yes
         1  sunny    warm     high  strong  warm     same   yes
         2  rainy    warm     high  strong  cool   change    no
         3  sunny    cold     high  strong  warm   change   yes
         4  sunny    warm   normal    weak  warm     same    no
```

```
Out[38]:    Sky_sunny  AirTemp_warm  Humidity_normal  Wind_weak  Water_warm  \
         0          1             1                1          0           1
         1          1             1                0          0           1
         2          0             1                0          0           0
```

```
3          1              0                0          0            1
4          1              1                1          1            1

   Forecast_same  Date?_yes
0              1          1
1              1          1
2              0          0
3              0          1
4              1          0
```

```
In [10]: # Complete these functions first
         # pos and neg are the number of positive and negative samples in a node
         def entropy(pos, neg):
             if pos <= 0 or neg <= 0:
                 return 0
             p = pos/(pos+neg)
             return - p*np.log2(p) - (1 - p)*np.log2((1 - p))

         # pos1 and pos2 are the number of positive examples in each branch after the split.
         # Same for neg1 and neg2
         def info_gain(pos1, neg1, pos2, neg2):
             pos, neg = pos1 + neg1, pos2 + neg2
             E_before = entropy(pos1+pos2, neg1+neg2)
             E_after = (pos * entropy(pos1, neg1) + neg * entropy(pos2, neg2))/ (pos+neg)
             return E_before - E_after
```

What is the class entropy for the entire dataset? What is the information gain when you split the data using the Water feature?

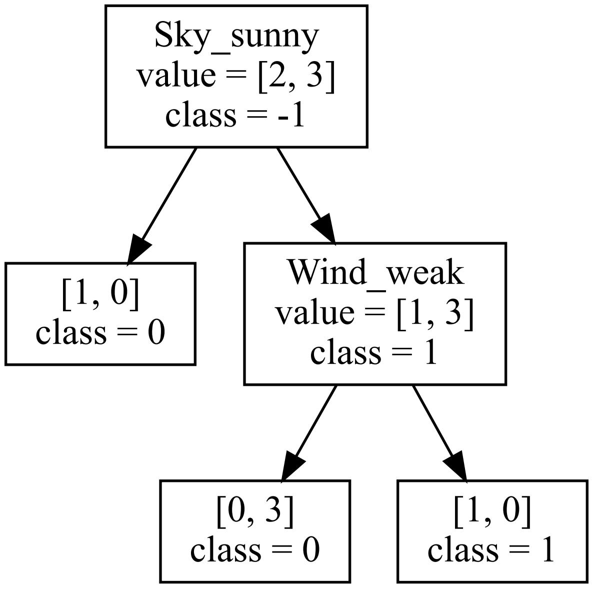
```
In [11]: pos1 = len(df1[df1['Date?_yes']==1])
         neg1 = len(df1[df1['Date?_yes']==0])
         E = entropy(pos1, neg1)   # class entropy of entire dataset
         print('Class entropy:', E)

         pos2 = len(df1[(df1['Water_warm']==0) & (df1['Date?_yes']==1)])
         neg2 = len(df1[(df1['Water_warm']==0) & (df1['Date?_yes']==0)])
         pos3 = len(df1[(df1['Water_warm']==1) & (df1['Date?_yes']==1)])
         neg3 = len(df1[(df1['Water_warm']==1) & (df1['Date?_yes']==0)])

         IG = info_gain(pos2, neg2, pos3, neg3)
         print('Information gain:', IG)
```

```
Class entropy: 0.970950594455
Information gain: 0.321928094887
```

Implement a basic decision tree

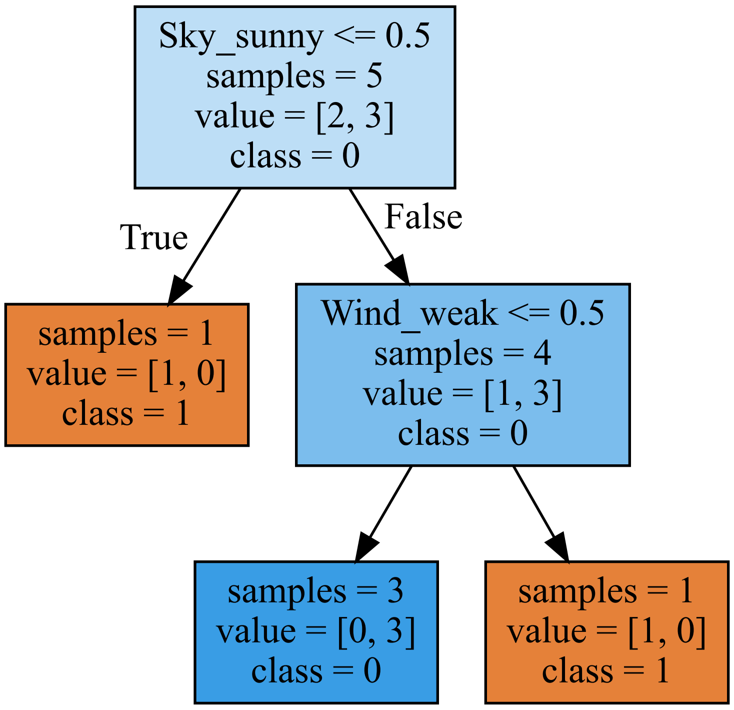
```
In [12]: from graphviz import Digraph

         def create_tree(df, features=None, dot=Digraph(), parent=None, sclass=-1):
             global i
             if features is None:
                 features = list(df)
                 features.remove('Date?_yes')
             max_IG, best_f = -1, -1
             node_id = str(i)
             i+=1

             # stop when count of either class is zero
             pos1 = len(df[df['Date?_yes']==1])
             neg1 = len(df[df['Date?_yes']==0])
             E = entropy(pos1, neg1)  # class entropy of entire dataset
             if E == 0:
                 # display nodes
                 dot.node(node_id, '[%d, %d] \n class = %d' % (neg1, pos1, sclass))
                 if parent:
                     dot.edge(parent, node_id)
                 return

             # find the best splitting feature
             for f in features:
                 pos2 = len(df[(df[f]==0) & (df['Date?_yes']==1)])
                 neg2 = len(df[(df[f]==0) & (df['Date?_yes']==0)])
                 pos3 = len(df[(df[f]==1) & (df['Date?_yes']==1)])
                 neg3 = len(df[(df[f]==1) & (df['Date?_yes']==0)])

                 IG = info_gain(pos2, neg2, pos3, neg3)
                 if IG > max_IG:
                     max_IG = IG
                     best_f = f

             # stop if gain is zero
             if max_IG <= 0:
                 # display nodes
                 dot.node(node_id, '[%d, %d] \n class = %d' % (neg1, pos1, sclass))
                 if parent:
                     dot.edge(parent, node_id)
                 return

             # split on that feature / display nodes
             dot.node(node_id, '%s \n value = [%d, %d] \n class = %d' % (best_f, neg1, pos1, scl
             if parent:
                 dot.edge(parent, node_id)
             features.remove(best_f)
             create_tree(df[df[best_f] == 0], features, dot, node_id, sclass=0)
```

```
            create_tree(df[df[best_f] == 1], features, dot, node_id, sclass=1)

            return dot

        # Create the tree and display the graph
        i=0
        g = Digraph(name='Decision Tree', node_attr={'shape': 'rectangle'})
        display(create_tree(df1, dot=g))
```



Train a scikit-learn decision tree on the same data

```
In [37]: from sklearn.tree import DecisionTreeClassifier
         from sklearn.tree import export_graphviz
         import graphviz

         # build a tree model
         X, y = df1.drop('Date?_yes', axis=1), df1['Date?_yes']
         clf = DecisionTreeClassifier(criterion='entropy', random_state=10)
         _ = clf.fit(X, y)

         # Creates a .dot file
         export_graphviz(clf, out_file="tree.dot", class_names=['1', '0'],
                         feature_names=list(X), impurity=False, filled=True)

         # Open and display
         with open("tree.dot") as f:
             dot_graph = f.read()
         display(graphviz.Source(dot_graph))
```



We can see clearly that the results from both decision trees are the same.

1.4 Random Forests (4 points (1+1+2))

Study the effect of the number of trees in a RandomForest on the EEG-eye-state dataset (http://www.openml.org/d/1471). This dataset measures brain activity using 15 sensors, and you need to predict whether the person's eyes are open or closed.

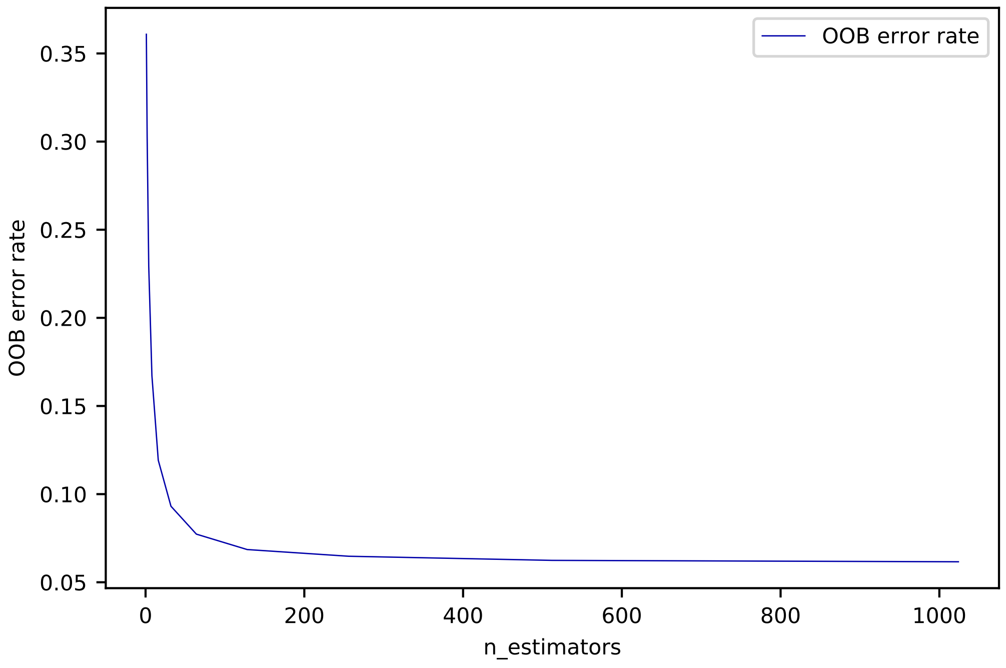
- Train a RandomForest classifier on this dataset with an increasing number of trees (on a log scale as above). Plot the Out-Of-Bag error against the number of trees.
  - The Out-Of-Bag error is the test error obtained when using bootstrapping, and using the non-drawn data points as the test set. This is what a RandomForest does internally, so you can retrieve it from the classifier. The code below hints on how to do this.
- Construct the same plot, but now use 10-fold Cross-validation and error rate instead of the OOB error. Compare the two. What do you learn from this?
- Compare the performance of the RandomForest ensemble with that of a single full decision tree. Compute the AUC as well as the bias and variance. Does the bias and variance increase/decrease for the ensemble? Does the number of trees affect the result?

Hint: Error rate = 1 - accuracy. It is not a standard scoring metric for cross_val_score, so you'll need to let it compute the accuracy values, and then compute the mean error rate yourself.
Hint: We discussed bias-variance decomposition in class. It is not included in scikit-learn, so you'll need to implement it yourself. Always first calculate the bias and variance of each sample individually, and then sum them up.

```
In [4]: from sklearn import ensemble
        eeg = oml.datasets.get_dataset(1471)  # Download Ionosphere data
        X, y = eeg.get_data(target=eeg.default_target_attribute)
```

Plot the Out-Of-Bag error against the number of trees.

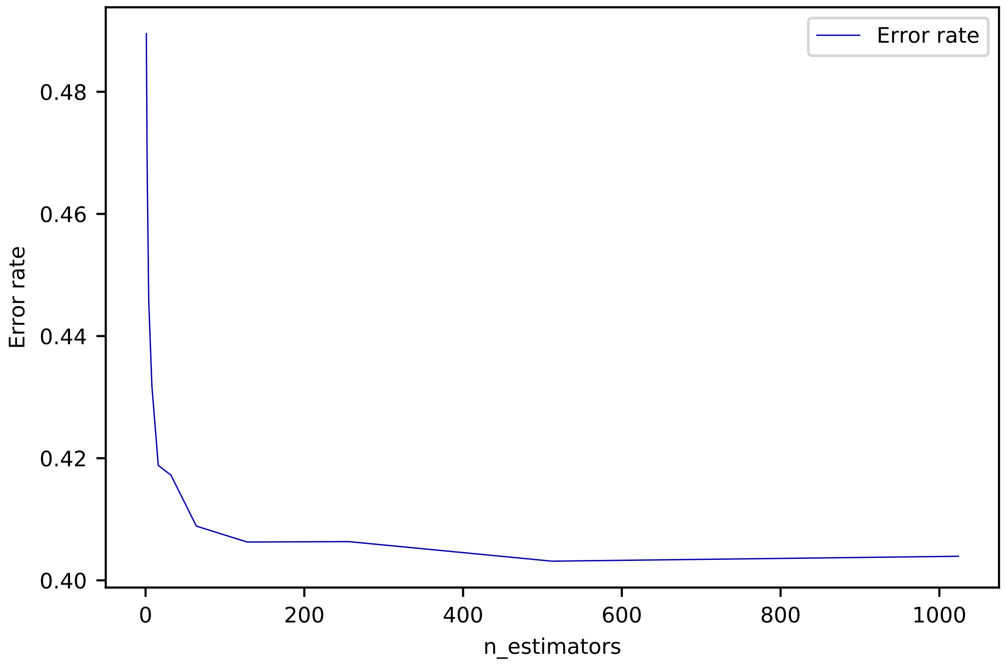
```
In [27]: # Out of bag errors can be retrieved from the RandomForest classifier. You'll need to l
         # http://scikit-learn.org/stable/auto_examples/ensemble/plot_ensemble_oob.html
         clf = ensemble.RandomForestClassifier(warm_start=True, oob_score=True, random_state=66)
         n_estimators = [2**i for i in range(0,11)]
         results = []

         for i in n_estimators:
             _ = clf.set_params(n_estimators=i)
             _ = clf.fit(X, y)
             results.append(1 - clf.oob_score_)

         # plot the data
         plt.plot(n_estimators, results)
         _ = plt.ylabel("OOB error rate")
         _ = plt.xlabel("n_estimators")

Out[27]: [<matplotlib.lines.Line2D at 0x1e3ffd7ae80>]
```



Construct the same plot, but now use 10-fold Cross-validation and error rate instead of the OOB error.

```
In [26]: from sklearn.model_selection import cross_val_score

         clf = ensemble.RandomForestClassifier(random_state=66)
         n_estimators = [2**i for i in range(0,11)]
         results = []

         for i in n_estimators:
             _ = clf.set_params(n_estimators=i)
             scores = cross_val_score(clf, X, y, cv=10, n_jobs=-1)
             results.append(1 - scores.mean())

         # plot the data
         plt.plot(n_estimators, results)
         _ = plt.ylabel("Error rate")
         _ = plt.xlabel("n_estimators")
```

```
Out[26]: [<matplotlib.lines.Line2D at 0x1e3f5c3b668>]
```



Both of them share the same trend, while the OOB error is much smaller than the CV error rate for fixed number of estimators. Comparing to the CV error, the OOB error plot is smoother. This is because the OOB error is taken as the average of the all estimated errors as the forest grows. Therefore OOB should be taken as the measure.

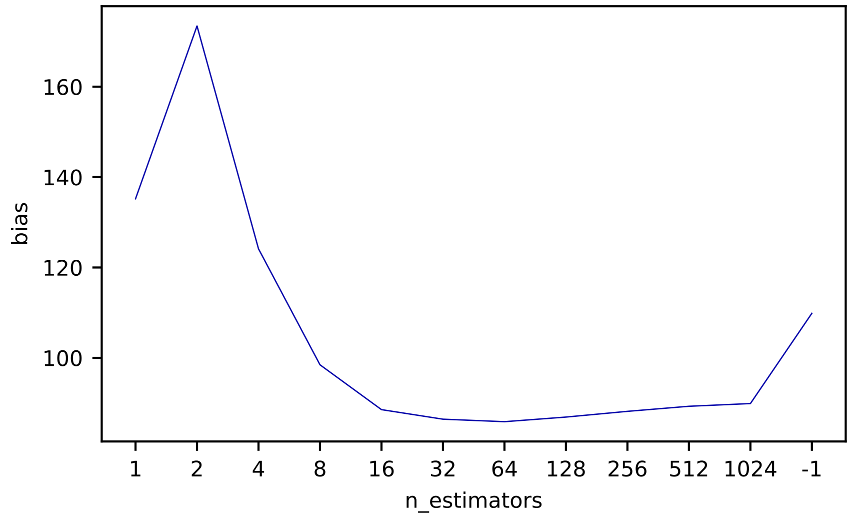
Compare the performance of the RandomForest ensemble with that of a single full decision tree.

```
In [23]: from sklearn.model_selection import cross_val_score, ShuffleSplit
         from sklearn.tree import DecisionTreeClassifier
         from sklearn.metrics import roc_auc_score

         n_repeat = 100
         ss = ShuffleSplit(n_splits=n_repeat, train_size=0.66, random_state=66)
         clf = ensemble.RandomForestClassifier(random_state=66, n_jobs=-1)
         n_estimators = [2**i for i in range(0, 11)]+[-1]
         results = {'bias': [], 'variance': [], 'score': []}

         def bias(x_true, x_predicted):
             return (1 - x_predicted.count(x_true)/len(x_predicted))**2

         def variance(x_predicted):
             P_class_1 = np.mean(x_predicted)
             P_class_0 = 1 - P_class_1
             return (1 - ((P_class_1)**2 + (P_class_0)**2))/ 2

         for c, i  in enumerate(n_estimators):
             predictions = [[] for _ in range(len(y))]
             scores = []

             if i == -1:
                 clf = DecisionTreeClassifier(random_state=66)
             else:
                 _ = clf.set_params(n_estimators=i)

             for train_index, test_index in ss.split(X):
                 # Train and predict
                 _ = clf.fit(X[train_index], y[train_index])
                 y_pred = clf.predict(X[test_index])
                 scores.append(roc_auc_score(y[test_index], y_pred))

                 # Store predictions
                 for k, index in enumerate(test_index):
                     predictions[index].append(y_pred[k])

             bias_sq = sum([bias(y[k], x) * (len(x) / n_repeat) for k, x in enumerate(prediction
             var = sum([variance(x) * (len(x) / n_repeat) for x in predictions])
             results['bias'].append(bias_sq)
             results['variance'].append(var)
             results['score'].append(np.mean(scores))
             print(results['bias'][c], results['variance'][c], results['score'][c])

         # plot the data
```

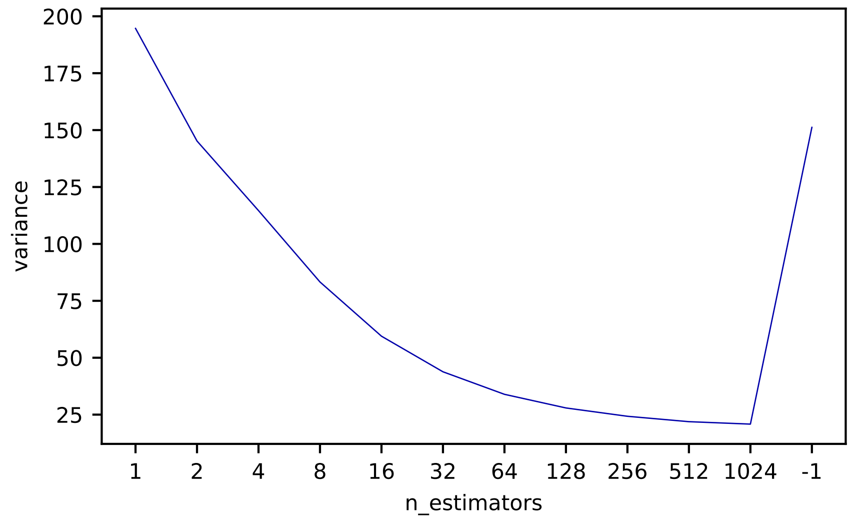
```
plt.plot(np.arange(len(n_estimators)), results['bias'])
_ = plt.ylabel("bias")
_ = plt.xlabel("n_estimators")
_ = plt.xticks(np.arange(len(n_estimators)), n_estimators)
plt.show()

plt.plot(np.arange(len(n_estimators)), results['variance'])
_ = plt.ylabel("variance")
_ = plt.xlabel("n_estimators")
_ = plt.xticks(np.arange(len(n_estimators)), n_estimators)
plt.show()

plt.plot(np.arange(len(n_estimators)), results['score'])
_ = plt.ylabel("roc_auc")
_ = plt.xlabel("n_estimators")
_ = plt.xticks(np.arange(len(n_estimators)), n_estimators)
plt.show()
```

```
135.198794955701 194.681205044 0.777363407561
173.45079915396448 145.199200846 0.770241303299
124.13527551553285 114.634724484 0.829863080466
98.4649823162248 83.2750176838 0.871913468914
88.54253331269976 59.4574666873 0.896368635433
86.4286539707653 43.8213460292 0.909118249689
85.875076187572 33.9249238124 0.91656016442
86.89602553062683 27.9339744694 0.920093762152
88.16297071057593 24.2770292894 0.921818292825
89.28486197681735 21.9051380232 0.922749457418
89.88569631306076 20.8343036869 0.923076251838
109.86830496592835 151.211695034 0.824075565527
```

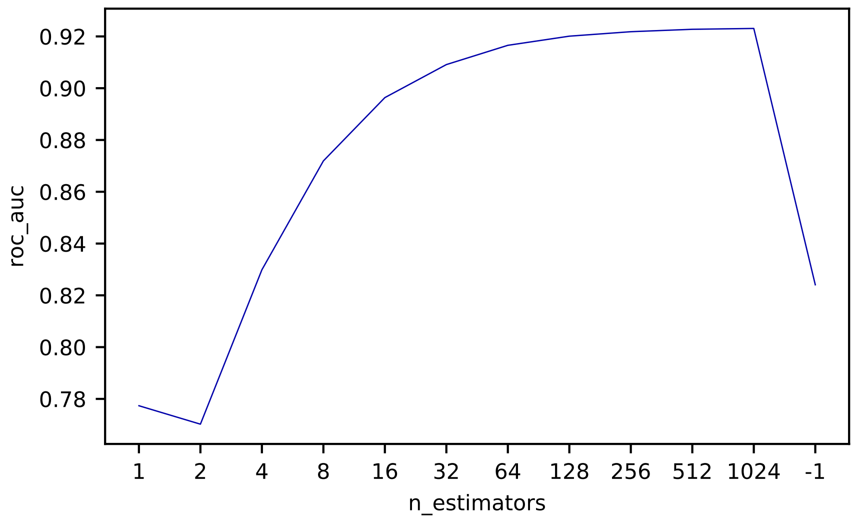
```
Out[23]: [<matplotlib.lines.Line2D at 0x24cb34b1dd8>]
```



```
Out[23]: [<matplotlib.lines.Line2D at 0x24ca6fc99b0>]
```



```
Out[23]: [<matplotlib.lines.Line2D at 0x24cb364eeb8>]
```



The *n_estimators* $= -1$ simply corresponds to a normal DecisionTreeClassifier as can seen from the code also. As the number of estimators grows, the bias and variance of the RF model decrees significantly became much smaller than of the single decision tree. As for the AUC score, the RF model outperforms single decision tree as nr- estimators grows

1.5 A regression benchmark (1 point)

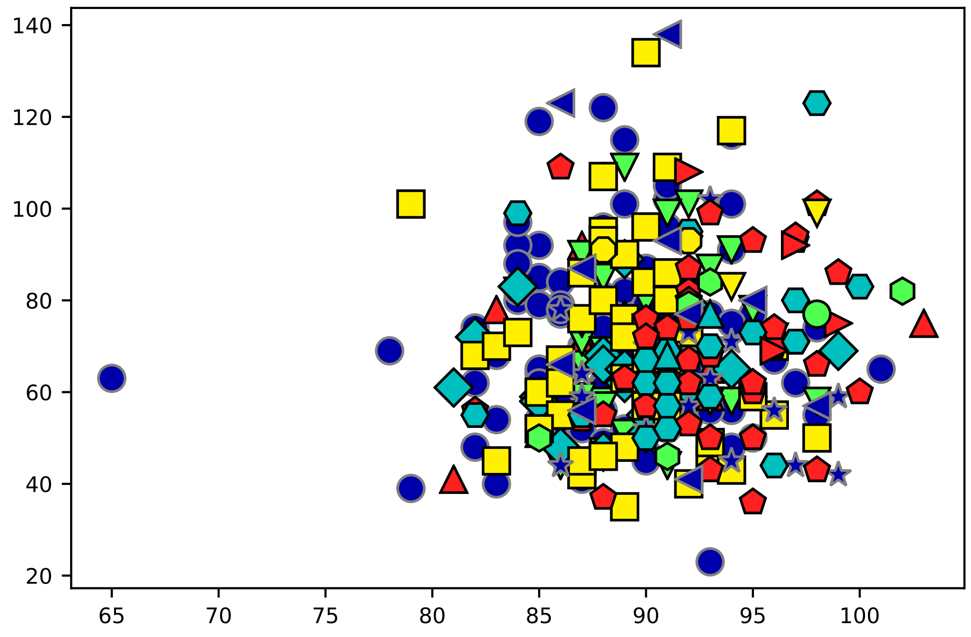
Consider the liver-disorder dataset (http://www.openml.org/d/8). The goal is to predict how much alcohol someone consumed based on blood test values.

- Take a selection of the algorithms that we covered in class that can do regression.
- Based on what you learned in the previous exercises, make educated guesses about good hyperparameter values and set up a grid or random search.
- Evaluate all models with 10-fold cross-validation and root mean squared error (RMSE). Report all results. Which model yields the best results?

Hint: mean squared error (MSE) is a standard scoring technique in GridSearchCV and cross_val_score. You'll have to compute the square roots yourself. Of course, during a grid search you can just use MSE, the optimal hyperparameter values will be the same.

```
In [34]: liver = oml.datasets.get_dataset(8)  # Download Liver-disorders data
         X, y = liver.get_data(target=liver.default_target_attribute)
         X.shape, y.shape
         _ = mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
         plt.show()
```

```
Out[34]: ((345, 5), (345,))
```



Take a selection of the algorithms that we covered in class that can do regression. Make educated guesses about good hyperparameter values.

```
In [35]: from sklearn.neighbors import KNeighborsRegressor
         from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet, SGDRegress
         from sklearn.ensemble import RandomForestRegressor
         from sklearn.tree import DecisionTreeRegressor

         estimators = [("knn", KNeighborsRegressor(n_jobs=-1)),
                       ("linear", LinearRegression(n_jobs=-1)),
                       ('ridge', Ridge()),
                       ('lasso', Lasso()),
                       ('elasticnet', ElasticNet()),
                       ('rf', RandomForestRegressor(random_state=66, n_jobs=-1)),
                       ('decision', DecisionTreeRegressor(random_state=66))]
         #               ('sgd', SGDRegressor(n_iter=30000))]

         param_grid = {
             'knn': {'n_neighbors': range(1, 51)},
             'linear': {},
             'ridge': {'alpha': [0.001, 0.01, 0.1, 1, 10, 100]},
             'lasso': {
                 'alpha': [0.001, 0.01, 0.1, 1, 10, 100],
                 'max_iter': [10000, 1000, 100]
             },
             'elasticnet': {
                 'alpha': [0.001, 0.01, 0.1, 1, 10, 100],
                 'l1_ratio': [0.05, 0.25, 0.5, 0.75, 0.95, 1],
                 'max_iter': [10000, 1000, 100]
             },
         #     'sgd': {
         #         'penalty': ['l2', 'l1', 'elasticnet'],
         #         'alpha': [10, 100, 1000, 10000, 100000],#0.001, 0.01, 0.1, 1,
         #         'l1_ratio': [0.05, 0.25, 0.5, 0.65, 0.75, 0.95, 1]
         #     }
             'rf': {
                 'n_estimators': [2**i for i in range(0,12)],
                 'max_features': np.arange(1, X.shape[1]+1)
             },
             'decision': {'max_features': np.arange(1, X.shape[1]+1)}
         }
```

Evaluate all models with 10-fold cross-validation and root mean squared error (RMSE).

```
In [36]: from sklearn.model_selection import GridSearchCV

         for key, clf in estimators:
             print('##### %s #####' % key)
             grid_search = GridSearchCV(clf, param_grid[key], cv=10, scoring='neg_mean_squared_e
             _ = grid_search.fit(X, y)
```

```
    # Get the RMSE
    scores = np.sqrt(-1*grid_search.cv_results_['mean_test_score'])
    scores_with_params = list(zip(grid_search.cv_results_['params'], scores))
    print('Scores:')
    scores_with_params
    print('Mean score:', np.mean(scores), '+-', np.var(scores))
```

```
##### knn #####
Scores:
```

```
Out[36]: [({'n_neighbors': 1}, 4.6718397828394904),
          ({'n_neighbors': 2}, 3.9625054290640813),
          ({'n_neighbors': 3}, 3.8609069212334477),
          ({'n_neighbors': 4}, 3.826987501068345),
          ({'n_neighbors': 5}, 3.7535374619498434),
          ({'n_neighbors': 6}, 3.6884836224622584),
          ({'n_neighbors': 7}, 3.6456876106424962),
          ({'n_neighbors': 8}, 3.5867042621002523),
          ({'n_neighbors': 9}, 3.5494412000129341),
          ({'n_neighbors': 10}, 3.51942436004625),
          ({'n_neighbors': 11}, 3.5437616308925772),
          ({'n_neighbors': 12}, 3.5179605880215297),
          ({'n_neighbors': 13}, 3.5083640734435151),
          ({'n_neighbors': 14}, 3.5361550010794596),
          ({'n_neighbors': 15}, 3.5437109473802924),
          ({'n_neighbors': 16}, 3.5329213211723527),
          ({'n_neighbors': 17}, 3.5394826479833688),
          ({'n_neighbors': 18}, 3.5266373827202897),
          ({'n_neighbors': 19}, 3.5459079805884355),
          ({'n_neighbors': 20}, 3.5381000796915067),
          ({'n_neighbors': 21}, 3.5371318492183939),
          ({'n_neighbors': 22}, 3.5335642313091804),
          ({'n_neighbors': 23}, 3.5324868865626216),
          ({'n_neighbors': 24}, 3.5318185044634176),
          ({'n_neighbors': 25}, 3.5168075114318911),
          ({'n_neighbors': 26}, 3.5234224295440333),
          ({'n_neighbors': 27}, 3.5250091276636897),
          ({'n_neighbors': 28}, 3.5315642666914977),
          ({'n_neighbors': 29}, 3.5284575843521484),
          ({'n_neighbors': 30}, 3.5268060208260144),
          ({'n_neighbors': 31}, 3.5267152379776698),
          ({'n_neighbors': 32}, 3.5279350703360808),
          ({'n_neighbors': 33}, 3.5257321867835962),
          ({'n_neighbors': 34}, 3.5271401470554102),
          ({'n_neighbors': 35}, 3.5331735020464876),
          ({'n_neighbors': 36}, 3.5341280160430859),
          ({'n_neighbors': 37}, 3.5297548629698645),
          ({'n_neighbors': 38}, 3.5271228740430485),
          ({'n_neighbors': 39}, 3.531601124178684),
          ({'n_neighbors': 40}, 3.5318927368123489),
          ({'n_neighbors': 41}, 3.5310696740451002),
          ({'n_neighbors': 42}, 3.5296395731178403),
          ({'n_neighbors': 43}, 3.5237484930850087),
          ({'n_neighbors': 44}, 3.5276483965901799),
          ({'n_neighbors': 45}, 3.5282790291439352),
          ({'n_neighbors': 46}, 3.5297407290806939),
          ({'n_neighbors': 47}, 3.5295789711200443),
          ({'n_neighbors': 48}, 3.5290144138972233),
          ({'n_neighbors': 49}, 3.5301921574352217),
          ({'n_neighbors': 50}, 3.5300451202108785)]

Mean score: 3.58539481065 +- 0.0325845161212
##### linear #####
Scores:


Out[36]: [({}, 3.4456431153534575)]

Mean score: 3.44564311535 +- 0.0
##### ridge #####
Scores:


Out[36]: [({'alpha': 0.001}, 3.4456429541410283),
          ({'alpha': 0.01}, 3.4456429901954428),
          ({'alpha': 0.1}, 3.4456433507597781),
          ({'alpha': 1}, 3.4456469584214031),
          ({'alpha': 10}, 3.44568323603474),
          ({'alpha': 100}, 3.4460653050861536)]

Mean score: 3.44572079911 +- 2.3943352142e-08
##### lasso #####
Scores:


Out[36]: [({'alpha': 0.001, 'max_iter': 10000}, 3.4456520755314606),
          ({'alpha': 0.001, 'max_iter': 1000}, 3.4456520755314606),
          ({'alpha': 0.001, 'max_iter': 100}, 3.4456520755314606),
          ({'alpha': 0.01, 'max_iter': 10000}, 3.4457314402092551),
          ({'alpha': 0.01, 'max_iter': 1000}, 3.4457314402092551),
          ({'alpha': 0.01, 'max_iter': 100}, 3.4457314402092551),
          ({'alpha': 0.1, 'max_iter': 10000}, 3.4466972827693763),
          ({'alpha': 0.1, 'max_iter': 1000}, 3.4466972827693763),
          ({'alpha': 0.1, 'max_iter': 100}, 3.4466972827693763),
          ({'alpha': 1, 'max_iter': 10000}, 3.4630936276346374),
          ({'alpha': 1, 'max_iter': 1000}, 3.4630936276346374),
          ({'alpha': 1, 'max_iter': 100}, 3.4630936276346374),
          ({'alpha': 10, 'max_iter': 10000}, 3.5743704593953392),
          ({'alpha': 10, 'max_iter': 1000}, 3.5743704593953392),
          ({'alpha': 10, 'max_iter': 100}, 3.5743704593953392),
          ({'alpha': 100, 'max_iter': 10000}, 3.7276453159068561),
          ({'alpha': 100, 'max_iter': 1000}, 3.7276453159068561),
          ({'alpha': 100, 'max_iter': 100}, 3.7276453159068561)]

Mean score: 3.51719836691 +- 0.0109467809334
##### elasticnet #####
Scores:
```

```
Out[36]: [({'alpha': 0.001, 'l1_ratio': 0.05, 'max_iter': 10000}, 3.4456450472610993),
          ({'alpha': 0.001, 'l1_ratio': 0.05, 'max_iter': 1000}, 3.4456450472610993),
          ({'alpha': 0.001, 'l1_ratio': 0.05, 'max_iter': 100}, 3.4456447414034912),
          ({'alpha': 0.001, 'l1_ratio': 0.25, 'max_iter': 10000}, 3.4456462971347173),
          ({'alpha': 0.001, 'l1_ratio': 0.25, 'max_iter': 1000}, 3.4456462971347173),
          ({'alpha': 0.001, 'l1_ratio': 0.25, 'max_iter': 100}, 3.4456462971347173),
          ({'alpha': 0.001, 'l1_ratio': 0.5, 'max_iter': 10000}, 3.4456482185289428),
          ({'alpha': 0.001, 'l1_ratio': 0.5, 'max_iter': 1000}, 3.4456482185289428),
          ({'alpha': 0.001, 'l1_ratio': 0.5, 'max_iter': 100}, 3.4456482185289428),
          ({'alpha': 0.001, 'l1_ratio': 0.75, 'max_iter': 10000}, 3.4456502169017842),
          ({'alpha': 0.001, 'l1_ratio': 0.75, 'max_iter': 1000}, 3.4456502169017842),
          ({'alpha': 0.001, 'l1_ratio': 0.75, 'max_iter': 100}, 3.4456502169017842),
          ({'alpha': 0.001, 'l1_ratio': 0.95, 'max_iter': 10000}, 3.4456517244213303),
          ({'alpha': 0.001, 'l1_ratio': 0.95, 'max_iter': 1000}, 3.4456517244213303),
          ({'alpha': 0.001, 'l1_ratio': 0.95, 'max_iter': 100}, 3.4456517244213303),
          ({'alpha': 0.001, 'l1_ratio': 1, 'max_iter': 10000}, 3.4456520755314606),
          ({'alpha': 0.001, 'l1_ratio': 1, 'max_iter': 1000}, 3.4456520755314606),
          ({'alpha': 0.001, 'l1_ratio': 1, 'max_iter': 100}, 3.4456520755314606),
          ({'alpha': 0.01, 'l1_ratio': 0.05, 'max_iter': 10000}, 3.4456594560271845),
          ({'alpha': 0.01, 'l1_ratio': 0.05, 'max_iter': 1000}, 3.4456594560271845),
          ({'alpha': 0.01, 'l1_ratio': 0.05, 'max_iter': 100}, 3.4456594560271845),
          ({'alpha': 0.01, 'l1_ratio': 0.25, 'max_iter': 10000}, 3.4456744400265733),
          ({'alpha': 0.01, 'l1_ratio': 0.25, 'max_iter': 1000}, 3.4456744400265733),
          ({'alpha': 0.01, 'l1_ratio': 0.25, 'max_iter': 100}, 3.4456744400265733),
          ({'alpha': 0.01, 'l1_ratio': 0.5, 'max_iter': 10000}, 3.4456935748379318),
          ({'alpha': 0.01, 'l1_ratio': 0.5, 'max_iter': 1000}, 3.4456935748379318),
          ({'alpha': 0.01, 'l1_ratio': 0.5, 'max_iter': 100}, 3.4456935748379318),
          ({'alpha': 0.01, 'l1_ratio': 0.75, 'max_iter': 10000}, 3.4457139971903579),
          ({'alpha': 0.01, 'l1_ratio': 0.75, 'max_iter': 1000}, 3.4457139971903579),
          ({'alpha': 0.01, 'l1_ratio': 0.75, 'max_iter': 100}, 3.4457139971903579),
          ({'alpha': 0.01, 'l1_ratio': 0.95, 'max_iter': 10000}, 3.4457274226921419),
          ({'alpha': 0.01, 'l1_ratio': 0.95, 'max_iter': 1000}, 3.4457274226921419),
          ({'alpha': 0.01, 'l1_ratio': 0.95, 'max_iter': 100}, 3.4457274226921419),
          ({'alpha': 0.01, 'l1_ratio': 1, 'max_iter': 10000}, 3.4457314402092551),
          ({'alpha': 0.01, 'l1_ratio': 1, 'max_iter': 1000}, 3.4457314402092551),
          ({'alpha': 0.01, 'l1_ratio': 1, 'max_iter': 100}, 3.4457314402092551),
          ({'alpha': 0.1, 'l1_ratio': 0.05, 'max_iter': 10000}, 3.4458089514240506),
          ({'alpha': 0.1, 'l1_ratio': 0.05, 'max_iter': 1000}, 3.4458089514240506),
          ({'alpha': 0.1, 'l1_ratio': 0.05, 'max_iter': 100}, 3.4458089514240506),
          ({'alpha': 0.1, 'l1_ratio': 0.25, 'max_iter': 10000}, 3.445971738339658),
          ({'alpha': 0.1, 'l1_ratio': 0.25, 'max_iter': 1000}, 3.445971738339658),
          ({'alpha': 0.1, 'l1_ratio': 0.25, 'max_iter': 100}, 3.445971738339658),
          ({'alpha': 0.1, 'l1_ratio': 0.5, 'max_iter': 10000}, 3.4461981583384893),
          ({'alpha': 0.1, 'l1_ratio': 0.5, 'max_iter': 1000}, 3.4461981583384893),
          ({'alpha': 0.1, 'l1_ratio': 0.5, 'max_iter': 100}, 3.4461981583384893),
          ({'alpha': 0.1, 'l1_ratio': 0.75, 'max_iter': 10000}, 3.4464407651125506),
          ({'alpha': 0.1, 'l1_ratio': 0.75, 'max_iter': 1000}, 3.4464407651125506),
          ({'alpha': 0.1, 'l1_ratio': 0.75, 'max_iter': 100}, 3.4464407651125506),
          ({'alpha': 0.1, 'l1_ratio': 0.95, 'max_iter': 10000}, 3.4466444294060548),
          ({'alpha': 0.1, 'l1_ratio': 0.95, 'max_iter': 1000}, 3.4466444294060548),
          ({'alpha': 0.1, 'l1_ratio': 0.95, 'max_iter': 100}, 3.4466444294060548),
          ({'alpha': 0.1, 'l1_ratio': 1, 'max_iter': 10000}, 3.4466972827693763),
          ({'alpha': 0.1, 'l1_ratio': 1, 'max_iter': 1000}, 3.4466972827693763),
          ({'alpha': 0.1, 'l1_ratio': 1, 'max_iter': 100}, 3.4466972827693763),
          ({'alpha': 1, 'l1_ratio': 0.05, 'max_iter': 10000}, 3.4476072397597992),
          ({'alpha': 1, 'l1_ratio': 0.05, 'max_iter': 1000}, 3.4476072397597992),
          ({'alpha': 1, 'l1_ratio': 0.05, 'max_iter': 100}, 3.4476072397597992),
          ({'alpha': 1, 'l1_ratio': 0.25, 'max_iter': 10000}, 3.4503664441943416),
          ({'alpha': 1, 'l1_ratio': 0.25, 'max_iter': 1000}, 3.4503664441943416),
          ({'alpha': 1, 'l1_ratio': 0.25, 'max_iter': 100}, 3.4503664441943416),
          ({'alpha': 1, 'l1_ratio': 0.5, 'max_iter': 10000}, 3.4548854637882185),
          ({'alpha': 1, 'l1_ratio': 0.5, 'max_iter': 1000}, 3.4548854637882185),
          ({'alpha': 1, 'l1_ratio': 0.5, 'max_iter': 100}, 3.4548854637882185),
          ({'alpha': 1, 'l1_ratio': 0.75, 'max_iter': 10000}, 3.4589898375252699),
          ({'alpha': 1, 'l1_ratio': 0.75, 'max_iter': 1000}, 3.4589898375252699),
          ({'alpha': 1, 'l1_ratio': 0.75, 'max_iter': 100}, 3.4589898375252699),
          ({'alpha': 1, 'l1_ratio': 0.95, 'max_iter': 10000}, 3.4621924601253578),
          ({'alpha': 1, 'l1_ratio': 0.95, 'max_iter': 1000}, 3.4621924601253578),
          ({'alpha': 1, 'l1_ratio': 0.95, 'max_iter': 100}, 3.4621924601253578),
          ({'alpha': 1, 'l1_ratio': 1, 'max_iter': 10000}, 3.4630936276346374),
          ({'alpha': 1, 'l1_ratio': 1, 'max_iter': 1000}, 3.4630936276346374),
          ({'alpha': 1, 'l1_ratio': 1, 'max_iter': 100}, 3.4630936276346374),
          ({'alpha': 10, 'l1_ratio': 0.05, 'max_iter': 10000}, 3.4755829584152114),
          ({'alpha': 10, 'l1_ratio': 0.05, 'max_iter': 1000}, 3.4755829584152114),
          ({'alpha': 10, 'l1_ratio': 0.05, 'max_iter': 100}, 3.4755829584152114),
          ({'alpha': 10, 'l1_ratio': 0.25, 'max_iter': 10000}, 3.5366428681956052),
          ({'alpha': 10, 'l1_ratio': 0.25, 'max_iter': 1000}, 3.5366428681956052),
          ({'alpha': 10, 'l1_ratio': 0.25, 'max_iter': 100}, 3.5366428681956052),
          ({'alpha': 10, 'l1_ratio': 0.5, 'max_iter': 10000}, 3.5618522899536713),
          ({'alpha': 10, 'l1_ratio': 0.5, 'max_iter': 1000}, 3.5618522899536713),
          ({'alpha': 10, 'l1_ratio': 0.5, 'max_iter': 100}, 3.5618522899536713),
          ({'alpha': 10, 'l1_ratio': 0.75, 'max_iter': 10000}, 3.5665982268973235),
          ({'alpha': 10, 'l1_ratio': 0.75, 'max_iter': 1000}, 3.5665982268973235),
          ({'alpha': 10, 'l1_ratio': 0.75, 'max_iter': 100}, 3.5665982268973235),
          ({'alpha': 10, 'l1_ratio': 0.95, 'max_iter': 10000}, 3.5727106495301242),
          ({'alpha': 10, 'l1_ratio': 0.95, 'max_iter': 1000}, 3.5727106495301242),
          ({'alpha': 10, 'l1_ratio': 0.95, 'max_iter': 100}, 3.5727106495301242),
          ({'alpha': 10, 'l1_ratio': 1, 'max_iter': 10000}, 3.5743704593953392),
          ({'alpha': 10, 'l1_ratio': 1, 'max_iter': 1000}, 3.5743704593953392),
          ({'alpha': 10, 'l1_ratio': 1, 'max_iter': 100}, 3.5743704593953392),
          ({'alpha': 100, 'l1_ratio': 0.05, 'max_iter': 10000}, 3.5634172160326232),
          ({'alpha': 100, 'l1_ratio': 0.05, 'max_iter': 1000}, 3.5634172160326232),
          ({'alpha': 100, 'l1_ratio': 0.05, 'max_iter': 100}, 3.5634172160326232),
          ({'alpha': 100, 'l1_ratio': 0.25, 'max_iter': 10000}, 3.6512253843827174),
          ({'alpha': 100, 'l1_ratio': 0.25, 'max_iter': 1000}, 3.6512253843827174),
          ({'alpha': 100, 'l1_ratio': 0.25, 'max_iter': 100}, 3.6512253843827174),
          ({'alpha': 100, 'l1_ratio': 0.5, 'max_iter': 10000}, 3.7281971036444057),
          ({'alpha': 100, 'l1_ratio': 0.5, 'max_iter': 1000}, 3.7281971036444057),
          ({'alpha': 100, 'l1_ratio': 0.5, 'max_iter': 100}, 3.7281971036444057),
          ({'alpha': 100, 'l1_ratio': 0.75, 'max_iter': 10000}, 3.7276453159068561),
          ({'alpha': 100, 'l1_ratio': 0.75, 'max_iter': 1000}, 3.7276453159068561),
          ({'alpha': 100, 'l1_ratio': 0.75, 'max_iter': 100}, 3.7276453159068561),
          ({'alpha': 100, 'l1_ratio': 0.95, 'max_iter': 10000}, 3.7276453159068561),
          ({'alpha': 100, 'l1_ratio': 0.95, 'max_iter': 1000}, 3.7276453159068561),
          ({'alpha': 100, 'l1_ratio': 0.95, 'max_iter': 100}, 3.7276453159068561),
          ({'alpha': 100, 'l1_ratio': 1, 'max_iter': 10000}, 3.7276453159068561),
          ({'alpha': 100, 'l1_ratio': 1, 'max_iter': 1000}, 3.7276453159068561),
          ({'alpha': 100, 'l1_ratio': 1, 'max_iter': 100}, 3.7276453159068561)]

Mean score: 3.50490342532 +- 0.0088617594994
##### rf #####
Scores:


Out[36]: [({'max_features': 1, 'n_estimators': 1}, 4.548291604738985),
          ({'max_features': 1, 'n_estimators': 2}, 3.7984550635179586),
          ({'max_features': 1, 'n_estimators': 4}, 3.6630499290357337),
          ({'max_features': 1, 'n_estimators': 8}, 3.6955773116884818),
          ({'max_features': 1, 'n_estimators': 16}, 3.4722600239488175),
          ({'max_features': 1, 'n_estimators': 32}, 3.4398785413919755),
          ({'max_features': 1, 'n_estimators': 64}, 3.4076287762624018),
          ({'max_features': 1, 'n_estimators': 128}, 3.3743953996668163),
          ({'max_features': 1, 'n_estimators': 256}, 3.3596206818971868),
          ({'max_features': 1, 'n_estimators': 512}, 3.3722451269895695),
          ({'max_features': 1, 'n_estimators': 1024}, 3.3678463463711039),
          ({'max_features': 1, 'n_estimators': 2048}, 3.3671248283237616),
          ({'max_features': 2, 'n_estimators': 1}, 4.3744979007947835),
          ({'max_features': 2, 'n_estimators': 2}, 3.9378321115977553),
          ({'max_features': 2, 'n_estimators': 4}, 3.6660160819401049),
          ({'max_features': 2, 'n_estimators': 8}, 3.5991960152958797),
          ({'max_features': 2, 'n_estimators': 16}, 3.5057293298339465),
          ({'max_features': 2, 'n_estimators': 32}, 3.4764684332250373),
          ({'max_features': 2, 'n_estimators': 64}, 3.4518075516519318),
          ({'max_features': 2, 'n_estimators': 128}, 3.4200848069334255),
          ({'max_features': 2, 'n_estimators': 256}, 3.4184291615371163),
          ({'max_features': 2, 'n_estimators': 512}, 3.4135205203366339),
          ({'max_features': 2, 'n_estimators': 1024}, 3.4147745746247131),
          ({'max_features': 2, 'n_estimators': 2048}, 3.4088097997437226),
          ({'max_features': 3, 'n_estimators': 1}, 4.3933414703801841),
          ({'max_features': 3, 'n_estimators': 2}, 3.9879347021241203),
          ({'max_features': 3, 'n_estimators': 4}, 3.8223456263080817),
          ({'max_features': 3, 'n_estimators': 8}, 3.6626666257788467),
          ({'max_features': 3, 'n_estimators': 16}, 3.5811724610317461),
          ({'max_features': 3, 'n_estimators': 32}, 3.4903893996561308),
          ({'max_features': 3, 'n_estimators': 64}, 3.4622447037743314),
          ({'max_features': 3, 'n_estimators': 128}, 3.4514790592505045),
          ({'max_features': 3, 'n_estimators': 256}, 3.4467242708964596),
          ({'max_features': 3, 'n_estimators': 512}, 3.4346004885684098),
          ({'max_features': 3, 'n_estimators': 1024}, 3.4390364149263686),
          ({'max_features': 3, 'n_estimators': 2048}, 3.4339381059226022),
          ({'max_features': 4, 'n_estimators': 1}, 4.4818474451657915),
          ({'max_features': 4, 'n_estimators': 2}, 4.0591639033430145),
          ({'max_features': 4, 'n_estimators': 4}, 3.686073477449876),
          ({'max_features': 4, 'n_estimators': 8}, 3.5755181037361461),
          ({'max_features': 4, 'n_estimators': 16}, 3.5483127113920059),
          ({'max_features': 4, 'n_estimators': 32}, 3.5256171407040338),
          ({'max_features': 4, 'n_estimators': 64}, 3.5005618321708658),
          ({'max_features': 4, 'n_estimators': 128}, 3.4809148023452963),
          ({'max_features': 4, 'n_estimators': 256}, 3.4770158651428291),
          ({'max_features': 4, 'n_estimators': 512}, 3.4611647628840525),
          ({'max_features': 4, 'n_estimators': 1024}, 3.4684154699327103),
          ({'max_features': 4, 'n_estimators': 2048}, 3.4637407578304562),
          ({'max_features': 5, 'n_estimators': 1}, 4.7067635577332538),
          ({'max_features': 5, 'n_estimators': 2}, 4.0150080765886962),
          ({'max_features': 5, 'n_estimators': 4}, 3.6935117416230381),
          ({'max_features': 5, 'n_estimators': 8}, 3.580461015980557),
          ({'max_features': 5, 'n_estimators': 16}, 3.5439671769712082),
          ({'max_features': 5, 'n_estimators': 32}, 3.5171789037697794),
          ({'max_features': 5, 'n_estimators': 64}, 3.4978281426269731),
          ({'max_features': 5, 'n_estimators': 128}, 3.4814112905533441),
          ({'max_features': 5, 'n_estimators': 256}, 3.4901823796668179),
          ({'max_features': 5, 'n_estimators': 512}, 3.4841913573646859),
          ({'max_features': 5, 'n_estimators': 1024}, 3.4958285582734954),
          ({'max_features': 5, 'n_estimators': 2048}, 3.4906089969932541)]

Mean score: 3.62141167867 +- 0.0964402772581
##### decision #####
Scores:
```

```
Out[36]: [({'max_features': 1}, 4.7800915642191253),
          ({'max_features': 2}, 4.6603476887148823),
          ({'max_features': 3}, 4.7539457296018854),
          ({'max_features': 4}, 4.8060951639181768),
          ({'max_features': 5}, 4.5342496942553874)]

Mean score: 4.70694596814 +- 0.00987704429714
```

We stopped testing with SGD regressor because it took too much time to train and also produced very poor results. Overall, Linear regression and Ridge regression scored the same and the best. ElasticNet and Lasso regression models produced the same $RMSE = 3.446$ as Linear and Ridge regression for some combination of the parameters but got progressively worse for other combinations especially from increase in alpha (greater regularization). You can see for which combinations the models produced the best result. Random Forests Regression even produced $RMSE \leq 3.446$ when $n\_estimators$ was sufficiently large and $max\_features$ generally low but on average was worse. Lastly, KNN regression and Decision tree regression produced relatively bad results.