TDT4173: Machine Learning and Case-Based Reasoning

Assignment 3 - Even L. Wanvik - 07.10.2019

1 Theory [1.9 points]

1.1) [0.5 points] What is the core idea of deep learning? How does it differ from shallow learning? Support your arguments with relevant examples.

At a basic level, **deep learning** is a machine-learning algorithm that teaches the application to filter inputs (in most cases images, text, or sound) through hidden computational layers, whose weights and connections (for sparsely connected layers) are updated to minimize a loss function through backpropagation. A more accurate representation of the field would be "representation learning" or "features learning", as it can learn representations and features directly from the input with little to no prior knowledge. As to what the core idea is, I would say that it is related to deep learning's ability to learn features directly from the data without the need for manual feature extraction.

I would also like to add that "Deep learning" is more a buzzword or marketing word rather than how the actual machine learning algorithm works, which might create confusion about how it works. A deep learning algorithm has multiple hidden layers between the input and output. These hidden layers are, for the most part, a large number of non-linear processing units, or neurons, in a crude analogy to how the brain processes information. Like natural signals, such as speech and vision, the layers is structured such that primitive features combine to form mid-level features while mid-level features combine to form high-level features. The term deep originates from the algorithm's ability to learn more complex features, the deeper the hidden layer is, i.e., the number of layers. By contrast, the machine learning algorithms without any form of layering can be called **Shallow learning**. With regards to neural networks, shallow learning also applies to NN's that consist of 1 or 2 hidden layers.

Let's have a look at support vector machines (SVM): A SVM are based on finding a splitting boundary, e.g. a hyperplane when the data is linearly separable, that is as far away from the support vectors (the outer part of a cluster "supporting" the boundary). The only "hidden" layer an SVM need is one to hold the support vector coefficients and is, therefore, a shallow model.

1.2) [0.8 points] Describe the comparison between following machine learning techniques: k-NN, decision tree, SVM and deep learning. Also, discuss the situations, for each of these techniques, as to why and when NOT to prefer them.

k-NN (k nearest neighbor):
The basic idea of k-NN is to explore the neighborhood of a datapoint. Assuming that the datapoint is similar to the neighboring data, we can predict the outcome of our datapoint. A k-NN classification algorithm will do a majority vote of the k nearest data points, while a k-NN regressor will instead compute the mean. As a rule of thumb, k should be an odd number. There is no training involved in the k-NN algorithm, it is a so-called lazy learning model, but the value of k can (and should) be optimized using a hold-out training data set. Compared to the other methods, it is an easy and straightforward learning model with few hyperparameters to tune, e.g., the number of neighboring points k. The algorithm should be avoided when dealing with datasets with many features, but can outperform most algorithms when training data is much larger than no. of features (m>>n). However, when classifying multi-class data, a k-NN algorithm is more acceptable. In short, k-NN is less computationally intensive, easy to implement, works best on heterogeneously distributed data, and it scales poorly with increased dimensionality.

Decision tree:
A decision tree, like the k-NN algorithm, uses a non-parametric approach that can be used for both classification or regression tasks. The goal is to learn a model that predicts the output by applying decision rules, in the form of If-then-else statements, inferred from the data features. The root node relates to all of the data, while the deeper the tree, the more complex the rules and fitter the model gets. Decision trees are so-called "white boxes" where the acquired knowledge can be expressed in readable form, while k-NN, SVM and NN are generally black boxes. Decision trees apply to both continuous and categorical inputs (CART (classification and regression tree).). A decision tree does not require preprocessing, just like the k-NN algorithm, nor any preliminary assumptions of the data. Also, like the k-NN algorithm, the decision tree is prone to outliers. Unlike the SVM, which uses non-linear kernel tricks or hyperplanes to map the data into different spaces, the decision tree derives hyper-rectangles in input space to solve the problem. Decision trees tend to work best when dealing with clearly defined problems with small to intermediate datasets and a manageable number of features.

SVM
As mentioned earlier that an SVM could perform a non-linear classification with ease using the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces. The non-linear property of the SVM allows us to capture much more complex relationships between the datapoints. The downside is that an SVM has a few parameters that require tuning and is much more computationally intensive when training the algorithm (O(N^2)-O(N^3)), i.e., it does not scale well with increased datasets. However, A SVM finds the best possible separation of the data, so with a small to an intermediate dataset and a larger set of features, the SVM is one of the go-to choices.

Deep learning
Unlike the "shallow networks" discussed earlier, where a lot of work might og into making sure that what we present for training in the input layer is already in a format that allows the network to recognise the important patterns, the idea with deep learning is that the additional layers can extract important features by itself. As mentioned in the previous task, the idea behind deep learning is additional layers, unlike the shallow methods mentioned above. Another key difference between deep learning vs. other machine learning algorithms stems from the way data is presented to the system. Machine learning algorithms almost always require structured data, whereas deep learning networks rely on "structured layers". Deep learning is most useful for problems with big datasets and a larger number of features and does not perform well for smaller datasets. It is also worth mentioning that deep learning in practice is hard and expensive.

1.3) [0.6 points] Discuss when should we use ensemble methods in context of machine learning? Explain briefly any 3 types of ensemble machine learning methods.

Ensemble methods is a machine learning technique that combines several base models in order to produce one optimal predictive model. Instead of using one single machine learning method to produce a predictive model, hoping that the model is the best possible predictor for our domain/data, ensemble methods combine different algorithms at different steps of the learning process and produce a more likely correct final model. Ensemble methods are using the predictions of small expert models in different parts of the input space. Most of the time, a single type of learning algorithm is used. However, some methods are heterogeneous, i.e., different weak learners are used.

We have three different types of machine learning methods: Decreasing the variance of our model (**bagging**), the bias (**boosting**), or improving predictions (**stacking**).

bagging:
Bagging (Bootstrap AGGregation) draws samples with replacement from a training set for approximating the sampling distribution, i.e., resampling from the sample data to create a large number of "phantom samples" known as bootstrap samples. These samples are then used to train multiple weak learners in parallel, whose predictions are aggregated and averaged into the most probable output with reduced variance. A normal bagging decision tree would be able to choose from all possible features at each node in all trees, which will

most likely cause the trees to be similar and hence choose the same outputs. A random forest tree, however, will only allow a random subset of features to be selected, further randomizing the tree and reducing the correlation between their outputs and its variance.

boosting:
In contrast to bagging, in which the learners are taught in parallel, boosting teaches the learners sequentially in a very adaptive way, i.e., a base model depends on the previous one and combines them following a deterministic strategy. This focus on reducing the error in poor predictions will, in turn, reduce the bias of the ensemble model. It is also worth mentioning that most boosting methods, as for the bagging methods, take a weighted average of the weak models, which means that a boosting method will also try to decrease the variance. Compared to bagging, the boosting methods can not teach the models in parallel and can be more computationally expensive if we are to fit several complex models in sequence. The two most important boosting algorithms are AdaBoost and gradient boosting. They differ by that boosting updates the weights attached to each of the training dataset observations, whereas gradient boosting updates the value of these observations.

stacking:
Stacking differs from boosting and bagging in that they, in most cases, use homogeneous weak learners, whereas stacking frequently uses heterogeneous weak learners. Another difference is that stacking learns how to combine the base models by training a meta-model, whereas bagging and boosting combine weak learners following deterministic algorithms. A good example in this assignment would be if we chose a k-NN, decision tree, and an SVM as weak learners, and use a neural network as a meta-model. The neural network will then take the outputs of the weak learners as inputs and learn to return final predictions based on them. It is normal to split out initial dataset into two, half for training the weak learners and the other portion for training the meta-model.
A possible extension of stacking is multi-level stacking. One example is a 3-level stacking method, where the first layer is the same as before, but the second layer is M meta-models whose outputs are inputs for a meta-model that produces a single output.

2 Programming [2.1 points]

In [1]:

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d, Axes3D
import math
import warnings
warnings.filterwarnings('ignore')

knn_regr = pd.read_csv("./dataset/knn_regression.csv")
knn_clas = pd.read_csv("./dataset/knn_classification.csv")

# Euclidean distance between two vectors
def euclidean_distance(row1, row2):
    dist = 0.0
    for i in range(len(row1)-1):
        dist += (row1[i] - row2[i])**2
    return math.sqrt(dist)

def get_neighbors(train, testrow, numNN):
    dist = list()
    for trainrow in train:
        dist.append((trainrow, euclidean_distance(testrow, trainrow)))
    dist.sort(key=lambda dist: dist[1])
    neighbors = list()
    for k in range(numNN):
        neighbors.append(dist[k][0])
    return neighbors
```

In [2]:

```
#------------ Regression ------------
k = 10  # Number of neighbors
testrow = knn_regr.values[123]                               # Extract the test vector from
 the training data
traindata = np.delete(knn_regr.values, obj=123, axis=0) # Remove our test vector from t
he training data
neighbors = get_neighbors(traindata, testrow, k)        # Get k closest neighbors
print("10 nearest neighbors:", neighbors)
regr_pred = np.round(np.mean(neighbors, 0)[3], 3)       # Extract avg of y column as pr
ediction
print("\nAverage of k neighbours:", regr_pred)
```

```
10 nearest neighbors: [array([6.2, 2.8, 4.8, 1.8]), array([6.3, 2.5, 4.9,
1.5]), array([6.3, 2.8, 5.1, 1.5]), array([6.3, 2.5, 5. , 1.9]), array([6.
1, 2.8, 4.7, 1.2]), array([6.1, 2.9, 4.7, 1.4]), array([6. , 2.7, 5.1, 1.
6]), array([6.1, 3. , 4.9, 1.8]), array([6.5, 2.8, 4.6, 1.5]), array([6.4,
2.7, 5.3, 1.9])]

Average of k neighbours: 1.61
```

In [3]:

```
#------------ Classification ------------
k = 10  # Number of neighbors
testrow = knn_clas.values[123]                        # Extract the test vector from
 the training data
traindata = np.delete(knn_clas.values, obj=123, axis=0) # Remove our test vector from t
he training data
neighbors = get_neighbors(traindata, testrow, k)        # Get k closest neighbors
print("10 nearest neighbors:", neighbors)
classes = [row[-1] for row in neighbors]
clas_pred = max(set(classes), key=classes.count)
print("\nMajority vote of k neighbours:", clas_pred)
```

```
10 nearest neighbors: [array([6.2, 2.8, 4.8, 1.8, 2. ]), array([6.3, 2.5,
5. , 1.9, 2. ]), array([6.1, 3. , 4.9, 1.8, 2. ]), array([6.3, 2.5, 4.9,
1.5, 1. ]), array([6.3, 2.8, 5.1, 1.5, 2. ]), array([6. , 2.7, 5.1, 1.6,
1. ]), array([6.4, 2.7, 5.3, 1.9, 2. ]), array([6. , 3. , 4.8, 1.8, 2. ]),
array([6.5, 2.8, 4.6, 1.5, 1. ]), array([6.5, 3. , 5.2, 2. , 2. ])]

Majority vote of k neighbours: 2.0
```

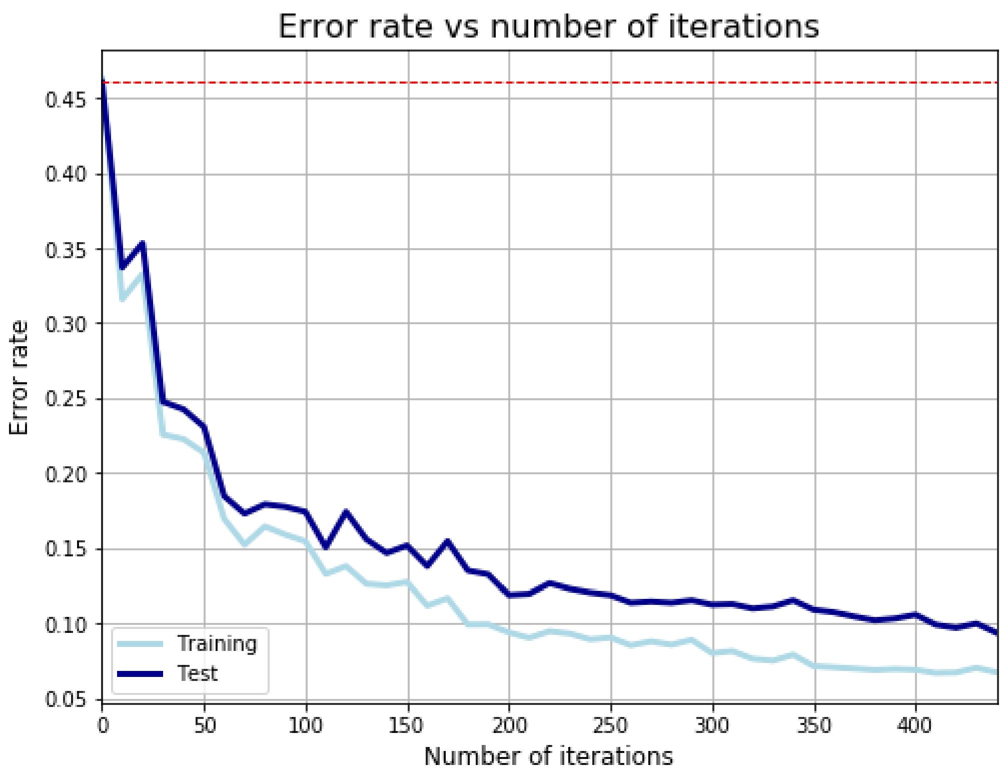
2.2) [0.7 points] AdaBoost implementation from scratch

I did this exercise together with Kristian Henriksen

In [4]:

```
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt

""" Get the error rate"""
def get_error_rate(pred, Y):
    return sum(pred != Y) / float(len(Y))

""" My AdaBoost implementation"""
def adaboost_clf(X_train, Y_train, X_test, Y_test, T, clf):
    n_train, n_test = len(X_train), len(X_test)
    # Initialize weights
    W = np.ones(n_train) / n_train
    pred_train, pred_test = [np.zeros(n_train), np.zeros(n_test)]

    for t in range(T):
        clf.fit(X_train, Y_train, sample_weight = W)
        train_pred_i = clf.predict(X_train)
        test_pred_i  = clf.predict(X_test)
        # miss1 for finding error, miss2 for quick calculation of weights
        miss1 = np.where(train_pred_i!=Y_train, 1, 0)
        miss2 = np.where(train_pred_i!=Y_train, 1, -1)
        # Calculate the error rate
        error = np.dot(W,miss1) / sum(W)
        # Calculate alpha (weight for classifier)
        alpha = 0.5 * np.log( (1 - error) / float(error))
        # New weights
        #W = np.multiply(W, np.exp([float(x) * alpha_m for x in miss2]))
        W *= np.exp(alpha*miss2)
        # Add predictions
        pred_train = [sum(x) for x in zip(pred_train, [x * alpha for x in train_pred_i
])]
        pred_test  = [sum(x) for x in zip(pred_test, [x * alpha for x in test_pred_i])]

        #hit = 1-miss1
        #hit_rate = sum(hit)/len(hit)
        #miss_rate = sum(miss1)/len(miss1)
        #print('The Accuracy of the {0}. model is : '.format(t+1),hit_rate*100,'%')
        #print('The missclassification rate is: ',miss_rate*100,'%')

    pred_train, pred_test = np.sign(pred_train), np.sign(pred_test)
    # Return error rate in train and test set
    return get_error_rate(pred_train, Y_train), get_error_rate(pred_test, Y_test)

def plot_error_rate(er_train, er_test):
    df_error = pd.DataFrame([er_train, er_test]).T
    df_error.columns = ['Training', 'Test']
    plot1 = df_error.plot(linewidth = 3, figsize = (8,6), color = ['lightblue', 'darkbl
ue'], grid = True)
    plot1.set_xlabel('Number of iterations', fontsize = 12)
    plot1.set_xticklabels(range(0,450,50))
    plot1.set_ylabel('Error rate', fontsize = 12)
    plot1.set_title('Error rate vs number of iterations', fontsize = 16)
    plt.axhline(y=er_test[0], linewidth=1, color = 'red', ls = 'dashed')

# Dataset
train = pd.read_csv("./dataset/adaboost_train.csv")
X_train = train[list(train)[2:]]
```

```
Y_train = train[list(train)[1]]
test  = pd.read_csv("./dataset/adaboost_test.csv")
X_test = test[list(test)[2:]]
Y_test = test[list(test)[1]]

# Get the error rate based on how many iterations we do
clf = DecisionTreeClassifier(max_depth = 1, random_state = 1)
er_train = list()
er_test = list()
iterations = range(1, 450, 10)
for i in iterations:
    er_i = adaboost_clf(X_train, Y_train, X_test, Y_test, i, clf)
    er_train.append(er_i[0])
    er_test.append(er_i[1])

# Compare error rate vs number of iterations
plot_error_rate(er_train, er_test)
```



The way I created the algorithm is a little inefficient. It has to do the iterations with exponential complexity, which might take a while when trying to simulate a high number of iterations. However, the error (miss) rate seems to quickly fall down to adequate values, and it will only continue to fall in an exponential manner the more iterations it runs.

Let's compare with a random forest classifier:

In [5]:

```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n_estimators=100,max_depth=500)
model.fit(X_train, Y_train)

test_pred = model.predict(X_test)
print(get_error_rate(test_pred, Y_test))
```

```
0.11
```

A random forest algorithm with 100 trees in paralell gives us the same result as ~400 stumps in paralell.

2.3) [0.7 points] k-NN vs SVM vs Random Forest on sklearn digit data set

In [6]:

```
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_digits

digits = load_digits()
X = digits.data
y = digits.target

X_train, X_test, y_train, y_test = train_test_split(X, y)

knn_clf = KNeighborsClassifier(n_neighbors=4)
svm_clf = SVC(gamma='scale', C=5)
rf_clf = RandomForestClassifier(n_estimators=66)


print('KNN accuracy: %f' % knn_clf.fit(X_train, y_train).score(X_test, y_test))
print('SVM accuracy: %f' % svm_clf.fit(X_train, y_train).score(X_test, y_test))
print('Random Forest accuracy: %f' % rf_clf.fit(X_train, y_train).score(X_test, y_test
))
```

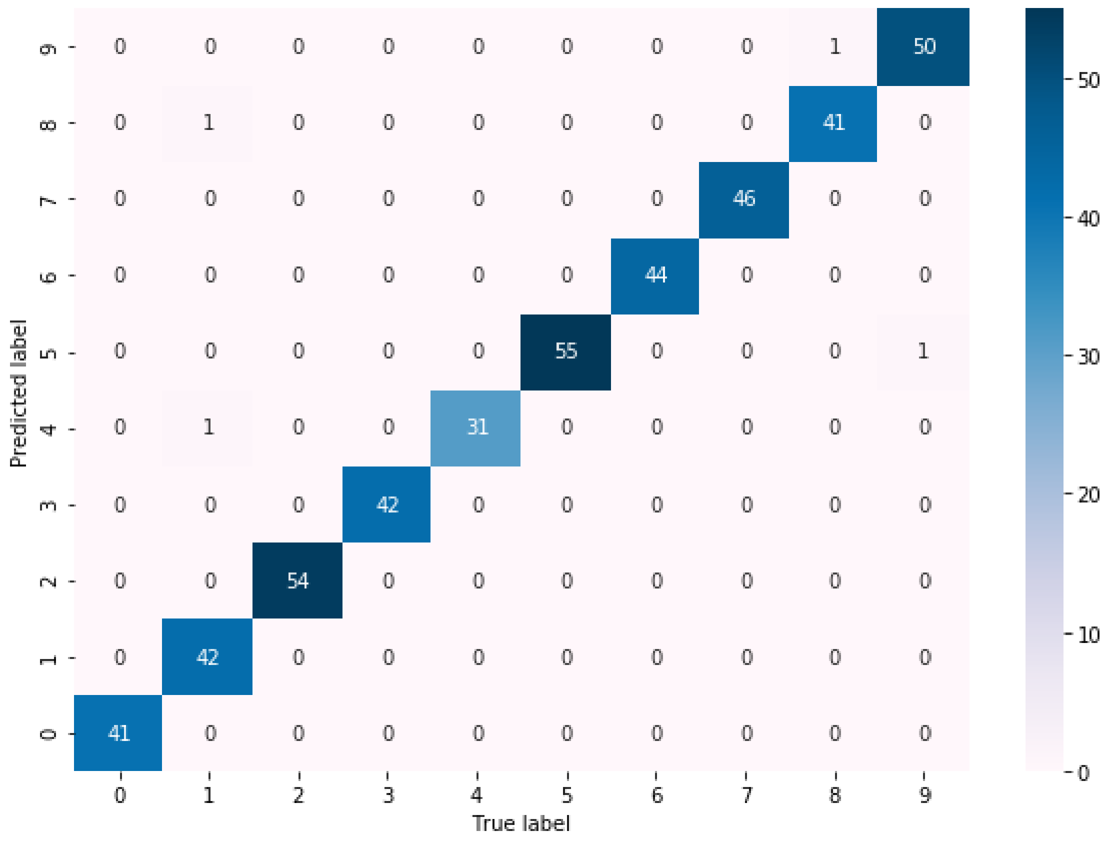
```
KNN accuracy: 0.991111
SVM accuracy: 0.986667
Random Forest accuracy: 0.982222
```

In [7]:

```
from sklearn.metrics import confusion_matrix
import seaborn as sn

y_pred = knn_clf.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
df_cm = pd.DataFrame(cm, index = [i for i in "0123456789"], columns = [i for i in "0123
456789"])
plt.figure(figsize = (10,7))
sn.heatmap(df_cm, annot=True, cmap="PuBu")
plt.xlabel("True label")
plt.ylabel("Predicted label")
plt.ylim(0,10)
```

Out[7]:

```
(0, 10)
```

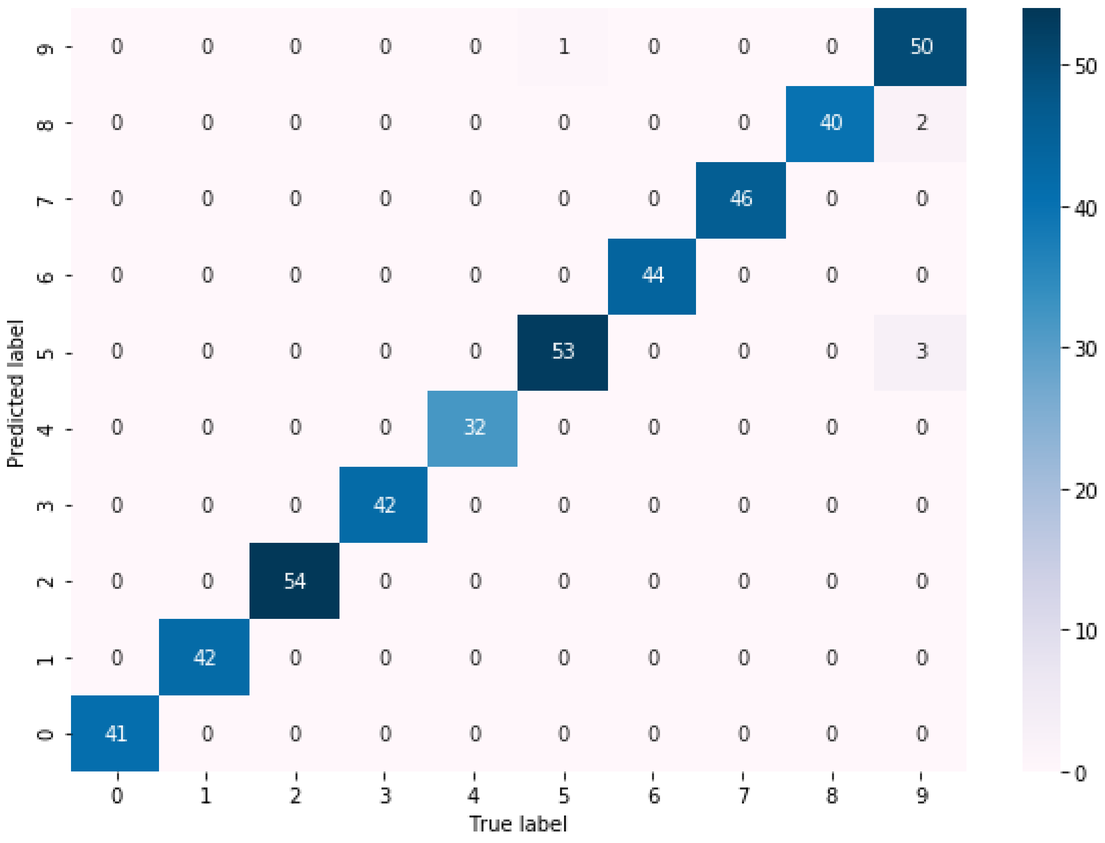


In [8]:

```
y_pred = svm_clf.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
df_cm = pd.DataFrame(cm, index = [i for i in "0123456789"], columns = [i for i in "0123
456789"])
plt.figure(figsize = (10,7))
sn.heatmap(df_cm, annot=True, cmap="PuBu")
plt.xlabel("True label")
plt.ylabel("Predicted label")
plt.ylim(0,10)
```

Out[8]:

```
(0, 10)
```

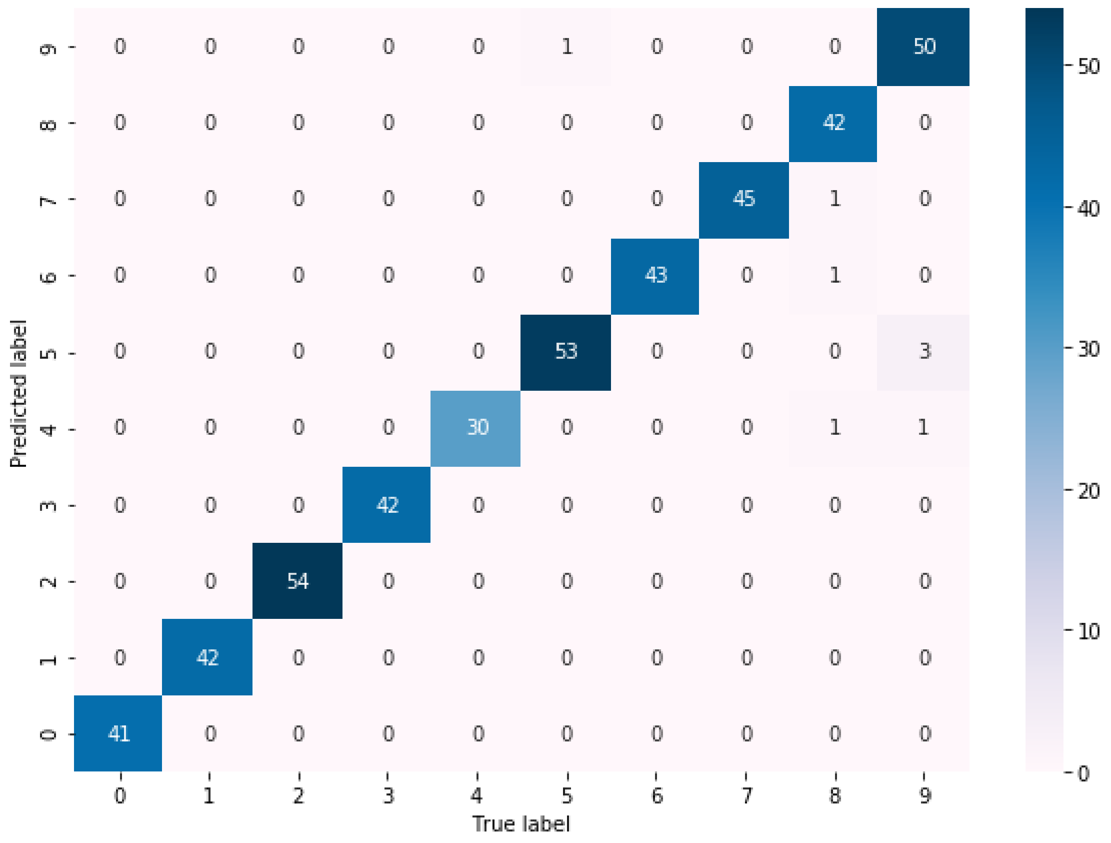


In [9]:

```
y_pred = rf_clf.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
df_cm = pd.DataFrame(cm, index = [i for i in "0123456789"], columns = [i for i in "0123
456789"])
plt.figure(figsize = (10,7))
sn.heatmap(df_cm, annot=True, cmap="PuBu")
plt.xlabel("True label")
plt.ylabel("Predicted label")
plt.ylim(0,10)
```

Out[9]:

```
(0, 10)
```



I tried to run parameter optimization for all of the classifiers and thought I would find the absolute best set of parameters. However, there seems to be some randomness as to how the classifiers are structured, as the optimal parameters changed every time. For the decision tree classifier, there seemed to be some form of periodicity as to what number of parallell trees gave the best score.