Fashion MNIST Image Classification

Rita Kurban

Minerva Schools at KGI

Author Note

First part: Problem Definition

Second part: Solution Specification

Third part: Testing and Analysis

Fourth part: Referencies

Fifth part: Appendices

Problem Definition

The purpose of this paper is to implement and analyze the performance of two image classification models. Image classification is one of the vital parts of digital image analysis and is widely used in social networks, recommendation systems, cloud apps, and more. To train these models, I will use Fashion MNIST dataset. Fashion MNIST consists of 70,000 images of clothing items which belong to 10 different categories. The authors intend this dataset to serve as a replacement for the original MNIST because they believe that MNIST is too easy and overused. Convolutional nets can score 99.7% accuracy on MNIST. Classic machine learning algorithms can also achieve 97% easily. It seems to be more challenging to get to the same level of accuracy on the fashion dataset, and that's what I'll try to do in this paper.

Solution Specifications

To classify images, I will use two machine learning techniques: Deep Convolutional Neural Networks and Random Forest. I also implemented a regular SVM from the scikit which we often used in our class. It has an accuracy of 76% and will serve as a benchmark for these two models.

CNN

Convolutional Neural Networks are deep, feed-forward neural networks that have successfully been applied to visual imagery analysis. A major advantage of CNNs is their independence from human efforts in feature design. They require relatively little pre-processing compared to other classification algorithms where the filters are hand-engineered.

A CNN consists of an input and an output layer, as well as multiple hidden layers. Convolutional layers apply a convolution operation to the input and pass the result to the next layer. Each neuron processes data only for its receptive field. This property is the main advantage over fully connected feedforward neural networks which can be used to classify images but are not practical because an enormous number of neurons would be necessary.

Pooling layers combine the outputs of neuron clusters at one layer into a single neuron in the next layer. For example, max pooling which I use in the model pools the maximum value from each of a cluster of neurons at the prior layer. To prevent overfitting, I also implement dropout layers. Dropout randomly turns off a fraction of neurons during the training process which reduces the dependency on the training set.

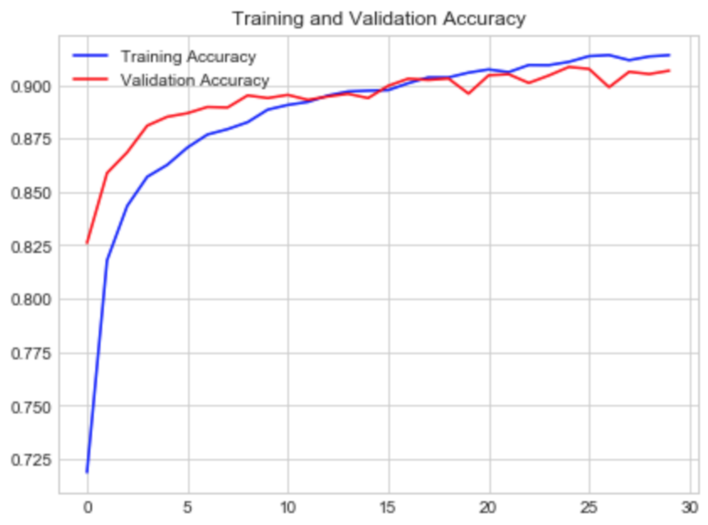
In the model, I also use advanced activation functions called Leaky ReLU and ELU. These functions are an attempt to fix the “dying ReLU” problem. Since ReLU returns 0 when the inputs are negative, the function gradient is also 0, so gradient descent learning will not alter the weights. Instead of the function being zero when $x < 0$, a leaky ReLU will have a small negative slope. As a result, the function won’t “die” and will further alter the weights. Exponential linear units, in turn, make the mean activations closer to 0 which speeds up learning. It has been shown that ELUs can obtain higher classification accuracy than ReLUs. Since no approach has been proven to have the best performance, I use a combination of different advanced activation functions to discover non-linear relationships and improve accuracy.

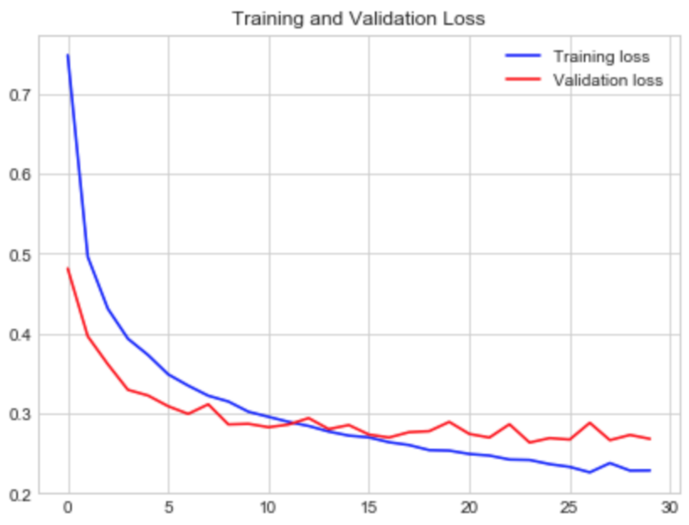
Random Forest

Random forest is another popular technique that has demonstrated excellent performance on a variety of tasks including image processing. Random forests operate by creating multiple decision trees and outputting the class that is the mode of the classes of the individual trees. This procedure corrects for decision trees' habit of overfitting to their training set. One crucial feature of Random Forest is bootstrap aggregating, or bagging. Bagging repeatedly selects a random sample from the training set with replacement and fits trees to these samples. This bootstrapping procedure leads to better performance since it decreases the variance without increasing the bias.

Testing and Analysis

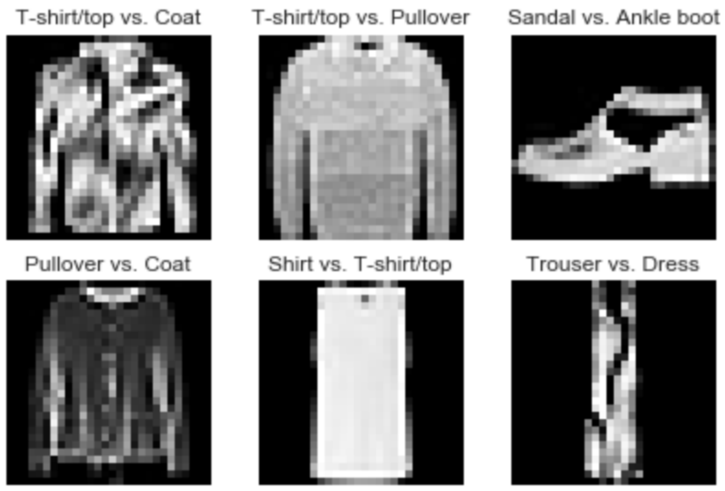
I trained the CNN and got an accuracy of almost 90% on the test set which is a 14% improvement over SVM. This is an impressive result given the fact that neural nets are known for their overfitting issue. To make sure overfitting wasn't a problem, I plotted validation loss and validation accuracy together with the training loss and training accuracy. As you can see on the graph, the validation loss decreases and not increases as it happens in case of overfitting, and there is no significant gap between training and validation accuracy.





Pic. 1. Training and Validation loss and accuracy of the CNN model

After that, I was curious why the model didn't manage to learn the data perfectly and plotted some of the incorrect examples. As you can see, they are pretty ambiguous even for a human being:



Pic. 2. Misclassified examples. Model's output is on the left, correct class is on the right.

My model learned a total of 438,026 parameters, and it took me almost all night to train it. Unfortunately, I have limited computational power and couldn't train a deeper model which would have a potential to classify all the images with high accuracy. Along with overfitting, this is one of the significant disadvantages of CNNs.

The second model, random forest, also had a surprisingly high accuracy on the test set: 88% (+12% over SVM). After reading the documentation on the random forest in Scikit-Learn, I found out that the most important settings are the number of trees in the forest *(n_estimators)* and the number of features considered for splitting at each leaf node *(max_features)*. I first tried to find optimal parameters with the help of *RandomizedSearchCV* but, unfortunately, my computational power wasn't enough for this task. As a result, I played around with this parameters and got the highest accuracy with *n_estimators = 100* and *max_features = auto.* Similar to the CNN, the accuracy of the model can be further optimized on a more powerful computer.

To sum up, CNNs have a potential to classify very ambiguous data given that they have enough training data and computational power. They are often used in industrial settings by huge corporations like Google and Facebook. Random forests are less promising but also have high performance on the fashion dataset. They are also trained much faster which is great for home usage. Both models are a significant improvement over a regular SVM model as implemented in scikit learn.

References

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

Hyperparameter Tuning the Random Forest in Python – Towards Data Science. (2018). Towards Data Science. Retrieved 18 April 2018, from https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74

zalandoresearch/fashion-mnist. (2018). GitHub. Retrieved 18 April 2018, from https://github.com/zalandoresearch/fashion-mnist

Appendix

Code

(The code in the notebook takes long to produce outputs, that's why I put them all here)

```
# Install all the necessary packages
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
from sklearn import metrics
from sklearn import svm
import keras
from keras.datasets import fashion_mnist
from keras.models import Sequential,Input,Model
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.utils import to_categorical #
from keras.layers.normalization import BatchNormalization
from keras.layers.advanced_activations import LeakyReLU, ELU

# Load the data
(train_X,train_Y), (test_X,test_Y) = fashion_mnist.load_data()
```

Data Exploration

In [39]:

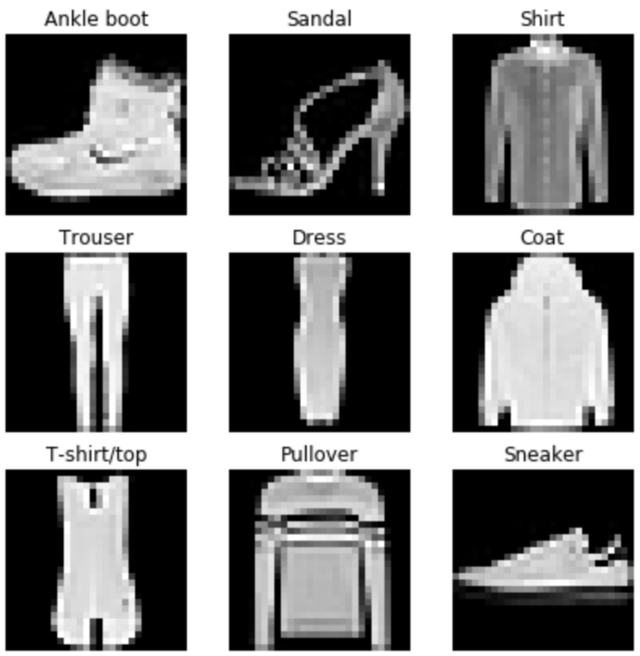
```
print 'Training set shape : ', train_X.shape, train_Y.shape
print 'Test set shape : ', test_X.shape, test_Y.shape
names = np.unique(train_Y)
print 'Labels:', names
Training set shape :  (60000, 28, 28) (60000,)
Test set shape :  (10000, 28, 28) (10000,)
Labels: [0 1 2 3 4 5 6 7 8 9]
```

In [40]:

```
# Dictionary that transforms digits into real labels
labels = {0: "T-shirt/top", 1: "Trouser", 2: "Pullover",
         3: "Dress", 4: "Coat", 5: "Sandal", 6: "Shirt",
         7: "Sneaker", 8: "Bag", 9: "Ankle boot"}
```

In [41]:

```
# Explore some of the examples (1 from each category)
plt.figure(figsize = [7,7])
index = [0,8000,3000,7900,59,457,4,5,6]
for i in range(1,10):
   plt.subplot(3,3,i)
   plt.axis("off")
   plt.imshow(train_X[index[i-1],:,:], cmap='gray')
   plt.title(labels[train_Y[index[i-1]]])
```



Data Pre-Processing

In [42]:

```
# Reshape the images to feed them to the CNN later
train_X = train_X.reshape(-1, 28,28, 1)
test_X = test_X.reshape(-1, 28,28, 1)
# Convert the data to float32 used by the network
train_X = train_X.astype('float32')
test_X = test_X.astype('float32')
# Rescale the pixels values to 0-1
train_X = train_X / 255.
test_X = test_X / 255.
```

In [43]:

```
# CNNs don't work with categorical variables
# Therefore, we need dummies
train_Y_dummy = to_categorical(train_Y)
test_Y_dummy = to_categorical(test_Y)
```

In [44]:

```
# Split the training set to get a validation set
train_X, valid_X, train_Y, valid_Y = train_test_split(train_X, train_Y_dummy, test_size=0.2)
train_X.shape,valid_X.shape,train_Y.shape,valid_Y.shape
```

Out[44]:

```
((48000, 28, 28, 1), (12000, 28, 28, 1), (48000, 10), (12000, 10))
```

Model 1: Convolutional Neural Network

In [53]:

```
batch_size = 64
epochs = 30
classes = 10
```

In [50]:

```
# Create a model
model = Sequential()
# Convolutional layers apply a convolution operation to the input.
```

```
model.add(Conv2D(64,(3, 3),activation='linear',input_shape=(28,28,1)))
# Advanced activation layer
model.add(LeakyReLU(alpha=0.1))
# Pooling layers combine the outputs at one layer into a single neuron in the next layer.
model.add(MaxPooling2D((2, 2)))
# Drop some neurons to avoid overfitting
model.add(Dropout(0.25))
model.add(Conv2D(128, (3, 3), activation='linear'))
model.add(ELU(alpha=0.1))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(256, (3, 3), activation='linear'))
model.add(LeakyReLU(alpha=0.1))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.4))
# Converting all the results into a single linear vector.
model.add(Flatten())
model.add(Dense(256, activation='linear'))
model.add(ELU(alpha=0.1))
model.add(Dropout(0.3))
# Classification lyer
model.add(Dense(classes, activation='softmax'))
```

In [51]:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_16 (Conv2D)           (None, 26, 26, 64)        640
_________________________________________________________________
leaky_re_lu_9 (LeakyReLU)    (None, 26, 26, 64)        0
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 13, 13, 64)        0
_________________________________________________________________
dropout_14 (Dropout)         (None, 13, 13, 64)        0
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 11, 11, 128)       73856
_________________________________________________________________
elu_6 (ELU)                  (None, 11, 11, 128)       0
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 5, 5, 128)         0
_________________________________________________________________
dropout_15 (Dropout)         (None, 5, 5, 128)         0
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 3, 3, 256)         295168
_________________________________________________________________
leaky_re_lu_10 (LeakyReLU)   (None, 3, 3, 256)         0
_________________________________________________________________
max_pooling2d_15 (MaxPooling (None, 1, 1, 256)         0
_________________________________________________________________
dropout_16 (Dropout)         (None, 1, 1, 256)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 256)               0
_________________________________________________________________
dense_3 (Dense)              (None, 256)               65792
_________________________________________________________________
elu_7 (ELU)                  (None, 256)               0
_________________________________________________________________
dropout_17 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                2570
```

```
=================================================================
Total params: 438,026
Trainable params: 438,026
Non-trainable params: 0
_________________________________________________________________
```

In [52]:

```
model.compile(loss=keras.losses.categorical_crossentropy,
optimizer=keras.optimizers.Adam(),metrics=['accuracy'])
```

In [54]:

```
train_model = model.fit(train_X, train_Y,
batch_size=batch_size,epochs=epochs,verbose=1,validation_data=(valid_X, valid_Y))
Train on 48000 samples, validate on 12000 samples
...
Epoch 30/30
48000/48000 [==============================] - 253s 5ms/step - loss: 0.2291 - acc: 0.9140 -
val_loss: 0.2686 - val_acc: 0.9067
```

In [55]:

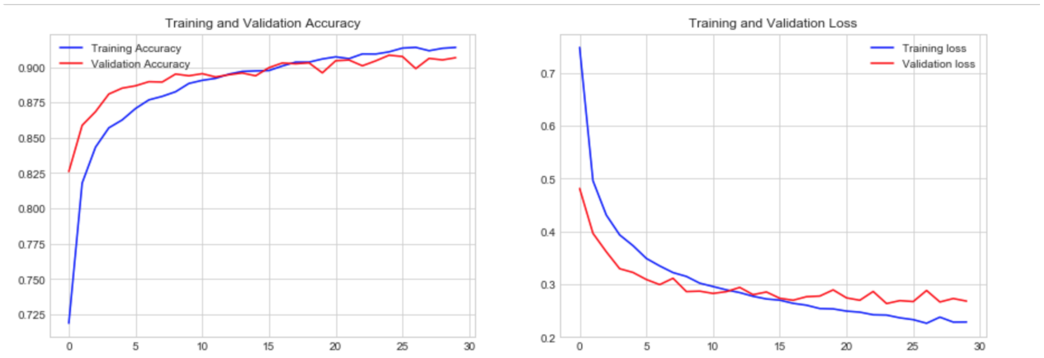
```
model.save("model.h5py")
```

In [56]:

```
evaluation = model.evaluate(test_X, test_Y_dummy, verbose=1)
print 'Test loss:', evaluation[0]
print'Test accuracy:', evaluation[1]
10000/10000 [==============================] - 11s 1ms/step
Test loss: 0.28907238229513166
Test accuracy: 0.8997
```

In [57]:

```
matplotlib.style.use('seaborn-whitegrid')
acc = train_model.history['acc']
val_acc = train_model.history['val_acc']
loss = train_model.history['loss']
val_loss = train_model.history['val_loss']
epochs = range(len(acc))

plt.figure(figsize = [15,5])
plt.subplot(1,2,1)
plt.plot(epochs, acc, 'b', label='Training Accuracy')
plt.plot(epochs, val_acc, 'b', c = "r", label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()
plt.subplot(1,2,2)
plt.plot(epochs, loss, 'b', label='Training loss')
plt.plot(epochs, val_loss, 'b', c = "r",label='Validation loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.show()
```

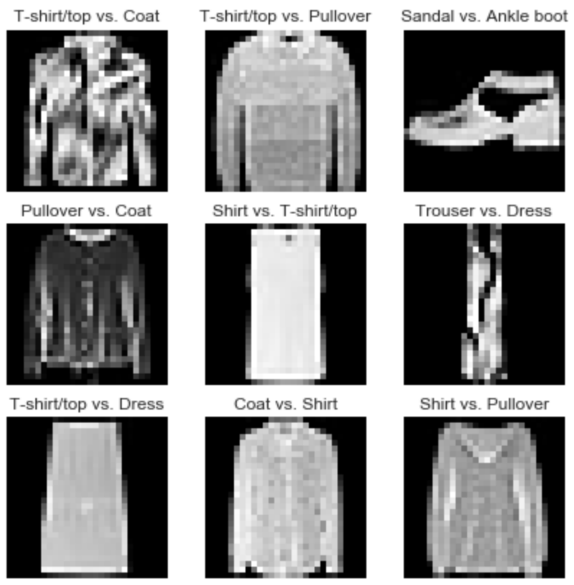


In [58]:

```
# Predict the labels of the test set
predict = np.argmax(np.round(model.predict(test_X)),axis=1)
```

In [59]:

```
# Have a look at incorrectly identified items
correct = np.where(predict==test_Y)[0]
incorrect = np.where(predict!=test_Y)[0]
print "Found %d correct labels." % len(correct)
print "Found %d incorrect labels." % len(incorrect)
matplotlib.style.use("seaborn-dark")
plt.figure(figsize = [7,7])
for i, incorrect in enumerate(incorrect[:9]):
   plt.subplot(3,3,i+1)
   plt.axis("off")
   plt.imshow(test_X[incorrect].reshape(28,28), cmap='gray', interpolation='none')
   #plt.title("Predicted {}, Class {}".format(predict[incorrect], test_Y[incorrect]))
   plt.title("{} vs. {}".format(labels[predict[incorrect]], labels[test_Y[incorrect]]))
Found 8904 correct labels.
Found 1096 incorrect labels.
```



In [61]:

```
target_names = ["Class: {}".format(labels[i]) for i in range(10)]
print(classification_report(test_Y, predict, target_names=target_names))
                   precision    recall  f1-score   support
```

```
Class: T-shirt/top       0.69      0.85      0.76      1000
    Class: Trouser       0.98      0.98      0.98      1000
   Class: Pullover       0.90      0.78      0.84      1000
      Class: Dress       0.92      0.89      0.91      1000
       Class: Coat       0.85      0.83      0.84      1000
     Class: Sandal       0.98      0.97      0.98      1000
      Class: Shirt       0.72      0.70      0.71      1000
    Class: Sneaker       0.96      0.96      0.96      1000
        Class: Bag       0.98      0.97      0.98      1000
 Class: Ankle boot       0.95      0.97      0.96      1000

       avg / total       0.90      0.89      0.89     10000
```

Model 2: Random Forest

In [70]:

```
rfc = RandomForestClassifier(n_jobs=-1, n_estimators=100)
```

In [65]:

```
(train_X,train_Y), (test_X,test_Y) = fashion_mnist.load_data()
```

In [66]:

```
# Data pre-processing (n_samples, n_features)
train_Y = train_Y.reshape(60000, -1)
train_X = train_X.reshape(60000, -1)
test_Y = test_Y.reshape(10000, -1)
test_X = test_X.reshape(10000, -1)
```

In [71]:

```
# Train Random Forest
rfc.fit(train_X, train_Y.ravel())
```

Out[71]:

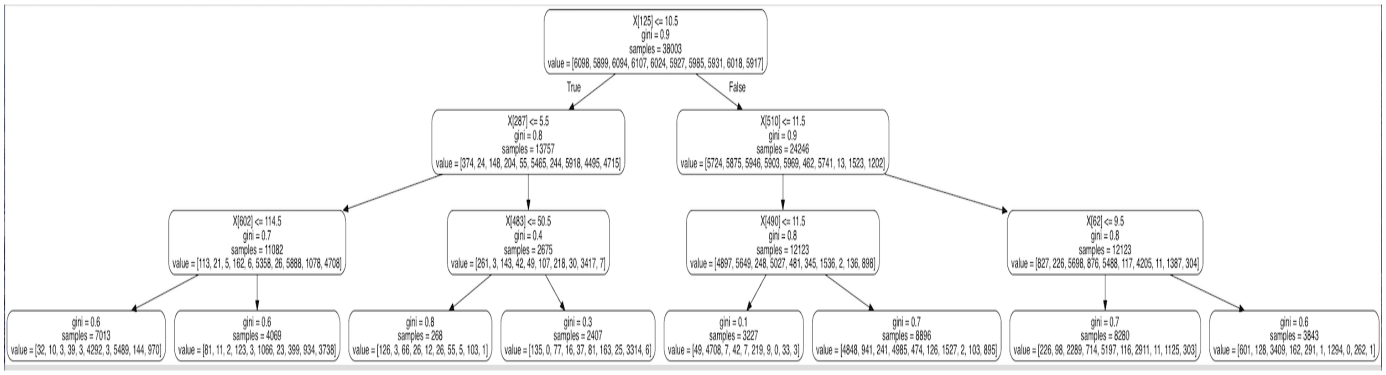
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=-1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

In [68]:

```
rf_pred = rfc.predict(test_X)

print("Accuracy: %f" % metrics.accuracy_score(test_Y, rf_pred)) # The fraction of correct
predictions
print("Classification report for classifier %s:\n%s\n"
     % (rfc, metrics.classification_report(test_Y, rf_pred)))
print("Confusion matrix:\n%s" % metrics.confusion_matrix(test_Y, rf_pred))
Accuracy: 0.879000
Classification report for classifier RandomForestClassifier(bootstrap=True, class_weight=None,
criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=1000, n_jobs=-1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False):
             precision    recall  f1-score   support

          0       0.83      0.86      0.84      1000
          1       0.99      0.96      0.98      1000
          2       0.77      0.80      0.79      1000
          3       0.88      0.91      0.89      1000
          4       0.78      0.83      0.80      1000
          5       0.98      0.96      0.97      1000
          6       0.72      0.60      0.66      1000
          7       0.93      0.95      0.94      1000
          8       0.96      0.97      0.97      1000
          9       0.95      0.95      0.95      1000

avg / total       0.88      0.88      0.88     10000


Confusion matrix:
[[857   0  10  31   3   1  85   0  13   0]
 [  2 962   3  23   3   0   5   0   2   0]
 [ 12   0 801  10 118   0  55   0   4   0]
 [ 18   2   6 912  27   0  33   0   2   0]
 [  1   0  91  33 826   0  47   0   2   0]
 [  0   0   0   1   0 960   0  27   1  11]
 [147   1 123  28  83   0 600   0  18   0]
 [  0   0   0   0   0  10   0 952   0  38]
 [  1   2   4   2   5   2   6   3 974   1]
 [  0   0   0   0   0   8   1  43   2 946]]
```

In [69]:

```
# Limit depth of tree to 3 levels
rf_small = RandomForestClassifier(n_estimators=100, max_depth = 3)
rf_small.fit(train_X, train_Y.ravel())
# Extract the small tree
tree_small = rf_small.estimators_[5]
# Save the tree as a png image
export_graphviz(tree_small, out_file = 'small_tree.dot', rounded = True, precision = 1)
(graph, ) = pydot.graph_from_dot_file('small_tree.dot')
graph.write_png('small_tree.png')
One of the trees:
```


X[125] <= 10.5
gini = 0.9
samples = 38003
value = [6098, 5899, 6094, 6107, 6024, 5927, 5985, 5931, 6018, 5917]
True
False
X[287] <= 5.5
gini = 0.8
samples = 13757
value = [374, 24, 148, 204, 55, 5465, 244, 5918, 4495, 4715]
X[510] <= 11.5
gini = 0.9
samples = 24246
value = [5724, 5875, 5946, 5903, 5969, 462, 5741, 13, 1523, 1202]
X[602] <= 114.5
gini = 0.7
samples = 11082
value = [113, 21, 5, 162, 6, 5358, 26, 5888, 1078, 4708]
X[483] <= 50.5
gini = 0.4
samples = 2675
value = [261, 3, 143, 42, 49, 107, 218, 30, 3417, 7]
X[490] <= 11.5
gini = 0.8
samples = 12123
value = [4897, 5649, 248, 5027, 481, 345, 1536, 2, 136, 898]
X[62] <= 9.5
gini = 0.8
samples = 12123
value = [827, 226, 5698, 876, 5488, 117, 4205, 11, 1387, 304]
gini = 0.6
samples = 7013
value = [32, 10, 3, 39, 3, 4292, 3, 5489, 144, 970]
gini = 0.6
samples = 4069
value = [81, 11, 2, 123, 3, 1066, 23, 399, 934, 3738]
gini = 0.8
samples = 268
value = [126, 3, 66, 26, 12, 26, 55, 5, 103, 1]
gini = 0.3
samples = 2407
value = [135, 0, 77, 16, 37, 81, 163, 25, 3314, 6]
gini = 0.1
samples = 3227
value = [49, 4708, 7, 42, 7, 219, 9, 0, 33, 3]
gini = 0.7
samples = 8896
value = [4848, 941, 241, 4985, 474, 126, 1527, 2, 103, 895]
gini = 0.7
samples = 8280
value = [226, 98, 2289, 714, 5197, 116, 2911, 11, 1125, 303]
gini = 0.6
samples = 3843
value = [601, 128, 3409, 162, 291, 1, 1294, 0, 262, 1]

Model 3: SVM (benchmark model)

In [77]:

```
clf = LinearSVC(random_state=0)
# Building a linear SVC
clf.fit(train_X, train_Y)
```

Out[77]:

```
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0)
```

In [79]:

```
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
    intercept_scaling=1, loss='squared_hinge', max_iter=1000,
    multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
    verbose=0)
# Predict on the original images
pred = clf.predict(test_X)

# Print performance metrics
print("Accuracy: %f" % metrics.accuracy_score(test_Y, pred))
print("Classification report for classifier %s:\n%s\n"
     % (clf, metrics.classification_report(test_Y, pred)))
print("Confusion matrix:\n%s" % metrics.confusion_matrix(test_Y, pred))
Accuracy: 0.760000
Classification report for classifier LinearSVC(C=1.0, class_weight=None, dual=True,
fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0):
             precision    recall  f1-score   support

          0       0.81      0.72      0.76      1000
          1       0.94      0.95      0.94      1000
          2       0.82      0.38      0.51      1000
          3       0.77      0.83      0.80      1000
          4       0.68      0.63      0.66      1000
          5       0.73      0.94      0.82      1000
          6       0.41      0.71      0.52      1000
          7       0.94      0.77      0.85      1000
          8       0.93      0.84      0.89      1000
          9       0.94      0.83      0.88      1000

avg / total       0.80      0.76      0.76     10000


Confusion matrix:
[[720   8   5  61   8   0 181   0  16   1]
 [  7 953   2  25   4   0   6   1   2   0]
 [ 13   8 375  29 176   0 393   0   6   0]
 [ 36  35   5 826  34   0  58   1   4   1]
 [  1   3  29  53 632   0 276   0   6   0]
 [  1   3   0   0   1 937   8  19   7  24]
 [107   5  35  58  67   0 709   0  19   0]
 [  1   0   0   0   0 194   3 773   0  29]
 [  4   4   7  15   4  19  96   5 845   1]
 [  1   0   0   1   0 131  15  22   0 830]]
```