# IST 718 Big Data Analytics

## Lab 3 (week 9)

**Ryan Timbrook**
**NetID:** RTIMBROO
**Course:** IST 718 Big Data Analytics
**Term:** Summer, 2019

**Assignment:** Identify Clothing Items

Lab 3 (week 9)

# Table of Contents

Lab 3 (week 9)

# 1 Introduction

Can we use algorithms and compute to identify clothing items? Specifically, can we determine which algorithm and compute methodology provides us the most efficient approach for classifying simple fashion images

## 1.1 About the Data

Using the base samples available from Zalando Research:

- Fashion-mnist

Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. It's intended that Fashion-MNIST serve as a direct **drop-in replacement** for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits.

- Each row is a separate image
- Column 1 is the class label.
- Remaining columns are pixel numbers (784 total).
- Each value is the darkness of the pixel (1 to 255)

### 1.1.1 *Labels*

Each training and test example is assigned to one of the following labels:

| Label | Description |
|---|---|
| 0 | T-shiert/top |
| 1 | Trouser |
| 2 | Pullover |
| 3 | Dress |
| 4 | Coat |
| 5 | Sandal |
| 6 | Shirt |
| 7 | Sneaker |
| 8 | Bag |

Lab 3 (week 9)

| Label | Description |
|---|---|
| 9 | Ankle boot |

### 1.1.2 *Dataset Info*

- Full training dataset has 6000 rows and 784 columns
- Full testing dataset has 1000 rows and 784 columns

## 1.2 Data Exploration & Cleaning

X_Train Dataset Info:
class: ndarray
shape: (48000, 784)
strides: (784, 1)
itemsize: 1
byteorder: little
type: uint8

X_Test Dataset Info:
class: ndarray
shape: (10000, 784)
strides: (784, 1)
itemsize: 1
byteorder: little
type: uint8

Sample Set of Training Images:

Sample Class 0: T-shiert/top Images:

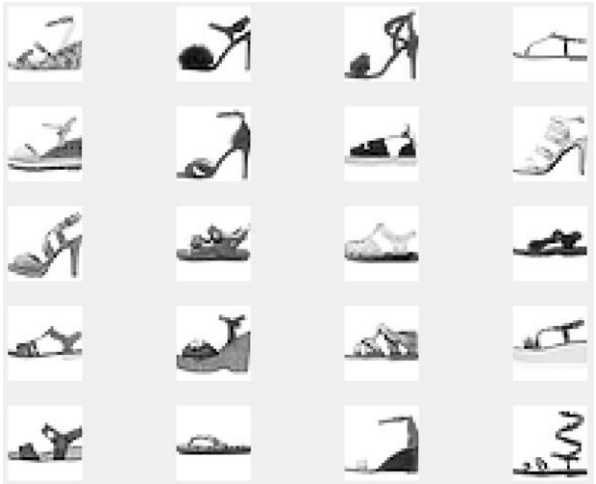
Sample Class 1: Trouser Images:

Sample Class 2: Pullover Images:

Sample Class 3: Dress Images:

Sample Class 4: Coat Images:

Sample Class 5: Sandle Images:



Sample Class 6: Shirt Images:

Sample Class 7: Sneaker Images:

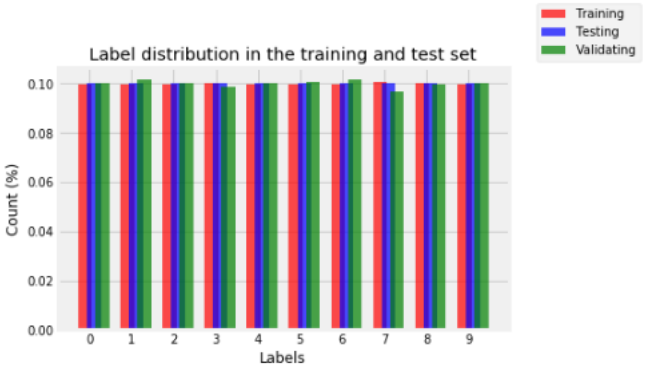
Sample Class 8: Bag Images:

Sample Class 9: Ankle Boot Images:

## 1.3 Data Transformation

The training dataset of 6000 sample images was split into training/validation (80/20 split) sets for model validation and prediction accuracy measurement. The test dataset of 1000 sample images was held out for prediction accuracy measurements.

The training/validation/testing datasets have nearly equal number of each class label. These datasets are considered balanced.



## 1.4 Data Normalization

The 'label' attribute was encoded to a catagorical, nominal variable for class classification training/testing. This was done using the sklearn.preprocessing.LabelEncoder class. Encode labels with value between 0 and n_classes-1.

Training and test dataset's were converted to float32 objects to minimizing memory computation needs and normalized from RGB color to black and white (0-1). This was accomplished by dividing the training and test data by 255.

# 2 Model - Random Forest Classifier

Python Package: scikit-learn v0.21.3 sklearn.ensemble.RandomForestClassifier

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement if bootstrap=True (default).

## 2.1 Analysis and Models

### 2.1.1 *Data Preprocessing*

Data preprocessing for this model is described above in section 1.4.

## 2.2 Model Baseline - Random Forest Classifier

This algorithm uses a perturb-and-combine techniques specifically designed for trees. A diverse set of classifieres are created by introducing randomness in the classifier construction. The prediction of the ensemble is given as the averaged prediction of the individual classifiers. Like decision trees, forests of trees also extend to multi-output problems. In random forests, each tree in the ensemble is built from a sample drawn with replacement. When splitting each node during the construction of a tree, the best split is found either from all input features or a random subset of size max_features.

### 2.2.1 *Model Details*

```
{'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': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 2,
 'warm_start': False}
```

### 2.2.2 *Model Parameters*

For this model, all default parameters were selected. Below is a listing of those parameters. For parameter tuning techniques, see this for more details.

| Parameter and Value | Description |
|---|---|
| **n_estimators=100** | The number of trees in the forest. |
| **criterion="gini"** | The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain. |
| **max_depth=None** | The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples |
| **min_samples_split=2** | The minimum number of samples required to split an internal node |
| **min_samples_leaf=1** | The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches |
| **min_weight_fraction_leaf=0.0** | The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided. |
| **max_features="auto"** | The number of features to consider when looking for the best split: If “auto”, then max_features=sqrt(n_features). |
| **max_leaf_nodes=None** | Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. |
| **min_impurity_decrease=0.0** | A node will be split if this split induces a decrease of the impurity greater than or equal to this value. |
| **min_impurity_split=None** | Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf. |
| **bootstrap=True** | Whether bootstrap samples are used when building trees. If False, the whole datset is used to build each tree. |
| **oob_score=False** | Whether to use out-of-bag samples to estimate the generalization accuracy. |
| **n_jobs=None** | The number of jobs to run in parallel for both fit and predict. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. |
| **random_state=None** | if int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random |
| **verbose=2** | Controls the verbosity when fitting and predicting. |
| **warm_start=False** | When set to True, reuse the solution of the previous call to fit and add more estimators to the |

| | |
|---|---|
| | ensemble, otherwise, just fit a whole new forest. |
| **class_weight=None** | Weights associated with classes in the form {class_label: weight}. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y. |

The main parameters to adjust when using these methods is n_estimators and max_features. The former is the number of trees in the forest. The larger the better, but also the longer it will take to compute. Also, the results will stop getting significantly better beyone a critical number of trees. The later is the size of the random subsets of features to consider when splitting a node. The lower the greater the reduction of variance, but also the greater the increase in bias.

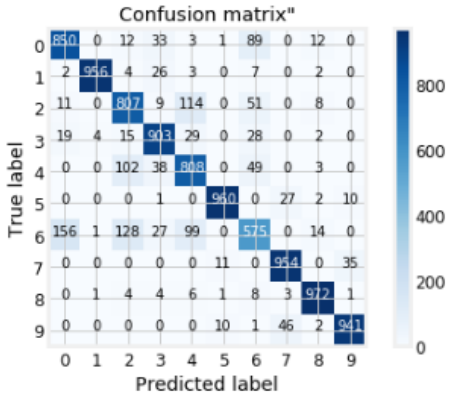
### 2.2.2.1 Model Results

| Fit Score | % Accuracy | Fit Time | Score Time | Predict Time | Precision | Recal | F1-Score |
|---|---|---|---|---|---|---|---|
| 0.8806 | 87% | 84.92204 | 0.0 | 0.5337 | 0.87 | 0.87 | 0.87 |

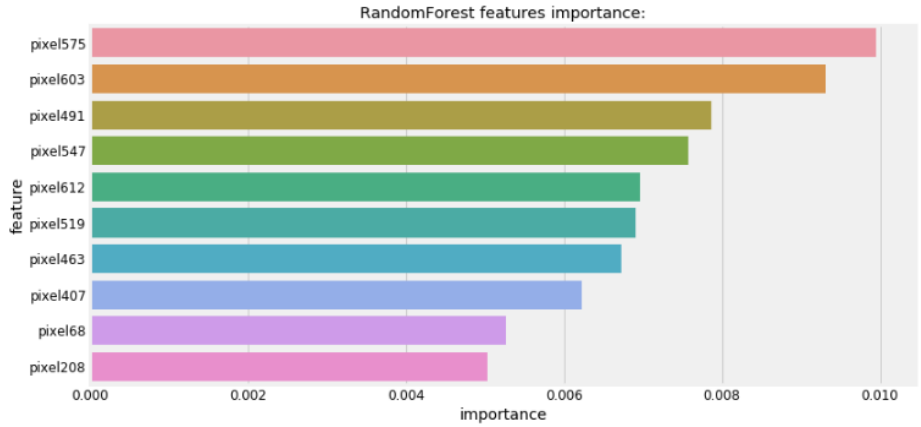
**Classification Report:**

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Class0 | 0.82 | 0.85 | 0.83 | 1000 |
| Class1 | 0.99 | 0.96 | 0.97 | 1000 |
| Class2 | 0.75 | 0.81 | 0.78 | 1000 |
| Class3 | 0.87 | 0.90 | 0.88 | 1000 |
| Class4 | 0.76 | 0.81 | 0.78 | 1000 |
| Class5 | 0.98 | 0.96 | 0.97 | 1000 |
| Class6 | 0.71 | 0.57 | 0.64 | 1000 |
| Class7 | 0.93 | 0.95 | 0.94 | 1000 |
| Class8 | 0.96 | 0.97 | 0.96 | 1000 |
| Class9 | 0.95 | 0.94 | 0.95 | 1000 |
| accuracy | | | 0.87 | 10000 |
| macro avg | 0.87 | 0.87 | 0.87 | 10000 |
| weighted avg | 0.87 | 0.87 | 0.87 | 10000 |

**Confusion Matrix Report:**




**Top 10 Feature Importance:**



## 2.3 Model Tuned - Random Forest Classification

### 2.3.1 *Model Details*

{'bootstrap': True,
'class_weight': None,
'criterion': 'gini',
'max_depth': 10,
'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': 200,
'n_jobs': 5,
'oob_score': False,
'random_state': 42,
'verbose': 0,
'warm_start': False}

### 2.3.2 *Model Parameters*

| Parameter and Value | Description |
|---|---|
| **n_estimators=200** | The number of trees in the forest. |
| **criterion="gini"** | The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain. |
| **max_depth=10** | The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples |
| **min_samples_split=2** | The minimum number of samples required to split an internal node |
| **min_samples_leaf=1** | The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches |
| **min_weight_fraction_leaf=0.0** | The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided. |
| **max_features="auto"** | The number of features to consider when looking for the best split: If “auto”, then max_features=sqrt(n_features). |
| **max_leaf_nodes=None** | Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. |
| **min_impurity_decrease=0.0** | A node will be split if this split induces a decrease of the impurity greater than or equal to this value. |

| | |
|---|---|
| **min_impurity_split=None** | Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf. |
| **bootstrap=True** | Whether bootstrap samples are used when building trees. If False, the whole datset is used to build each tree. |
| **oob_score=False** | Whether to use out-of-bag samples to estimate the generalization accuracy. |
| **n_jobs=5** | The number of jobs to run in parallel for both fit and predict. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. |
| **random_state=42** | if int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random |
| **verbose=0** | Controls the verbosity when fitting and predicting. |
| **warm_start=False** | When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. |
| **class_weight=None** | Weights associated with classes in the form {class_label: weight}. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y. |

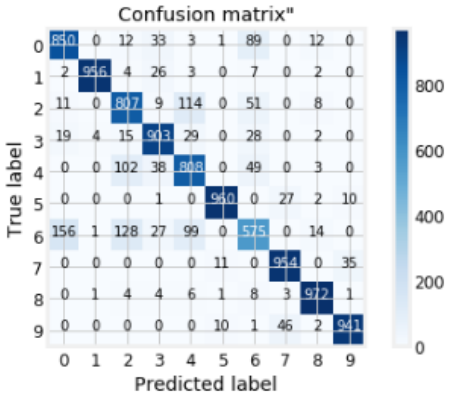
### 2.3.3 *Model Results*

| Fit Score | % Accuracy | Fit Time | Score Time | Predict Time | Precision | Recal | F1-Score |
|---|---|---|---|---|---|---|---|
| 0.851 | 84% | 25.9838 | 0.0 | 0.4462 | 0.85 | 0.85 | 0.85 |

**Classification Report:**

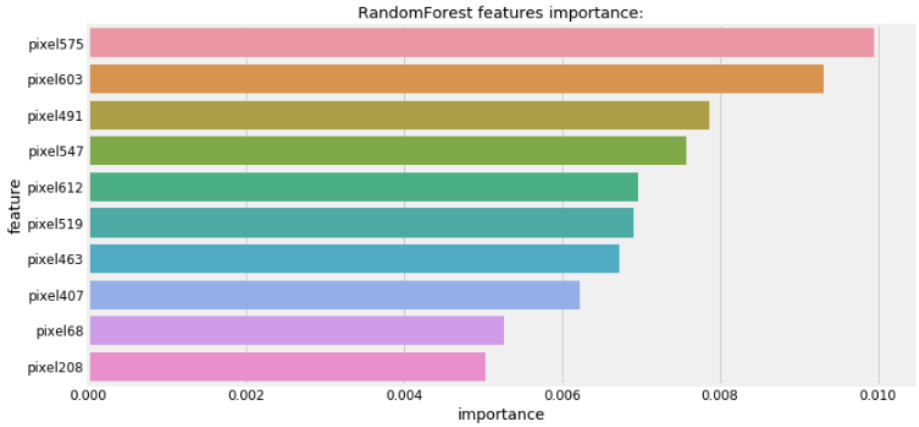
| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Class0 | 0.79 | 0.84 | 0.82 | 1202 |
| Class1 | 1.00 | 0.95 | 0.98 | 1219 |
| Class2 | 0.74 | 0.75 | 0.74 | 1205 |
| Class3 | 0.81 | 0.91 | 0.86 | 1184 |
| Class4 | 0.69 | 0.82 | 0.75 | 1202 |
| Class5 | 0.97 | 0.92 | 0.95 | 1211 |
| Class6 | 0.75 | 0.50 | 0.60 | 1218 |
| Class7 | 0.89 | 0.92 | 0.91 | 1159 |
| Class8 | 0.96 | 0.96 | 0.96 | 1197 |
| Class9 | 0.92 | 0.94 | 0.93 | 1203 |
| accuracy | | | 0.85 | 12000 |
| macro avg | 0.85 | 0.85 | 0.85 | 12000 |
| weighted avg | 0.85 | 0.85 | 0.85 | 12000 |

**Confusion Matrix Report:**



Lab 3 (week 9)

## Top 10 Feature Importance:



# 3 Model - XGBoosted Trees

XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting.

See documentation for more indepth details.

- XGBoost Documentation
- XGBoost Parameter details

## 3.1 Analysis and Models

### 3.1.1 *Data Preprocessing*

Data preprocessing for this model is described above in section 1.4.

## 3.2 Model Baseline - XGBoost

### 3.2.1 *Model Details*

{'base_score': 0.5,
'booster': 'gbtree',
'colsample_bylevel': 1,
'colsample_bynode': 1,
'colsample_bytree': 1,
'gamma': 0,
'learning_rate': 0.1,
'max_delta_step': 0,
'max_depth': 5,
'min_child_weight': 1,
'missing': nan,
'n_estimators': 50,
'nthread': 1,
'objective': 'multi:softprob',
'reg_alpha': 0,
'reg_lambda': 1,
'scale_pos_weight': 1,
'seed': 0,
'subsample': 1,
'verbosity': 1}

### 3.2.2 *Model Parameters*

All other default parameters used for this model.

| Parameter and Value | Description |
|---|---|
| **n_estimators=50** | int Number of boosted trees to fit |

| | |
|---|---|
| **max_depth=5** | The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples |

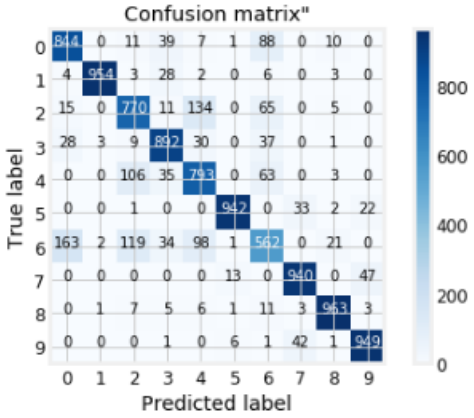
### 3.2.3 *Model Results*

| Fit Score | % Accuracy | Fit Time | Score Time | Predict Time | Precision | Recal | F1-Score |
|---|---|---|---|---|---|---|---|
| 0.8688 | **86%** | 654.5434 | 0.0 | 0.7612 | 0.86 | 0.86 | 0.86 |

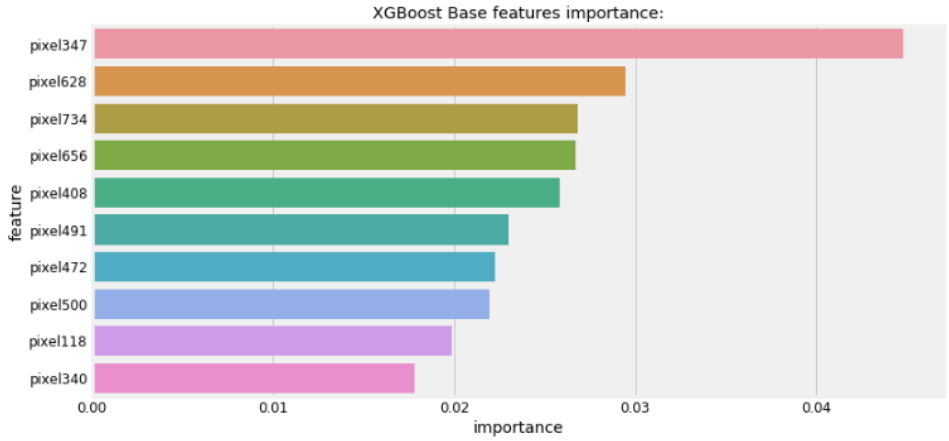
**Classification Report:**

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Class0 | 0.80 | 0.84 | 0.82 | 1000 |
| Class1 | 0.99 | 0.95 | 0.97 | 1000 |
| Class2 | 0.75 | 0.77 | 0.76 | 1000 |
| Class3 | 0.85 | 0.89 | 0.87 | 1000 |
| Class4 | 0.74 | 0.79 | 0.77 | 1000 |
| Class5 | 0.98 | 0.94 | 0.96 | 1000 |
| Class6 | 0.67 | 0.56 | 0.61 | 1000 |
| Class7 | 0.92 | 0.94 | 0.93 | 1000 |
| Class8 | 0.95 | 0.96 | 0.96 | 1000 |
| Class9 | 0.93 | 0.95 | 0.94 | 1000 |
| accuracy | | | 0.86 | 10000 |
| macro avg | 0.86 | 0.86 | 0.86 | 10000 |
| weighted avg | 0.86 | 0.86 | 0.86 | 10000 |

**Confusion Matrix Report:**




**Top 10 Feature Importance**



## 3.3 Model Tuned - XGBoost

### 3.3.1 *Model Details*

XGBoost with GridSearch, an exhaustive search over specified parameter values for an estimator. GridSearchCV implements a 'fit' a 'score', and 'predict' method. The parameters of the estimator used to apply these methods are optimized by cross-validated grid-search over a parameter grid.

Python package: scikit-learn v0.21.3 sklearn.model_selection.GridSearchCV

```
{'base_score': 0.5,
 'booster': 'gbtree',
 'colsample_bylevel': 1,
 'colsample_bynode': 1,
 'colsample_bytree': 0.8,
 'gamma': 0,
 'learning_rate': 0.1,
 'max_delta_step': 0,
 'max_depth': 10,
 'min_child_weight': 1,
 'missing': nan,
 'n_estimators': 100,
 'nthread': 1,
 'objective': 'multi:softprob',
 'reg_alpha': 0,
 'reg_lambda': 1,
 'scale_pos_weight': 1,
 'seed': 42,
 'subsample': 1,
 'verbosity': 1}
```

### 3.3.2 *Model Parameters*

Using GridSearch parameters specified below, all other parameters are set to default values.

```
param_grid = [{
   'max_depth':[5,10],
   'n_estimators':[100],
   'learning_rate':[0.05, 0.1],
   'colsample_bytree':[0.8, 0.95]
}]
```

| Parameter and Value | Description |
|---|---|
| **estimator** | This is assumed to implement the scikit-learn estimator interface. Either estimator needs to provide a score function, or scoring must be passed. |
| **param_grid** | Dictionary with parameters names (string) as keys and lists of parameter settings to try as values, or a list of such dictionaries, in which case the grids spanned by each dictionary in the list are explored. This enables searching over any sequence of parameter settings. |

| | |
|---|---|
| **scoring** | A single string or a callable to evaluate the predictions on the test set. |
| **n_jobs** | Number of jobs to run in parallel. None means 1. -1means using all processers |
| **pre_dispatch** | Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be: None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs. An int, giving the exact number of total jobs that are spawned. A string, giving an expression as a function of n_jobs, as in '2*n_jobs'. |
| **iid** | If True, return the average score across folds, weighted by the number of samples in each test set. If False, return the average score across folds. Default is True |
| **cv** | Determines the cross-validation splitting strategy.Possible inputs for cv are: None, to use the default 3-fold cross validation, integer, to specify the number of folds in a (Stratified)KFold |
| **refit** | Refit an estimator using the best found parameters on the whole dataset. For multiple metric evaluation, this needs to be a string denoting the scorer that would be used to find the best parameters for refitting the estimator at the end. |
| **verbose** | Controls the verbosity; the higher, the more messages |
| **error_score** | Value to assign to the score if an error in estimator fitting. If set to 'raise', the error is raised. If a numeric value is given, FitFailedWanring is raised. |
| **return_train_score** | If False, the cv_results_ attribute will not include training scores. Computing training scores is used to get insights on how different parameter settings impact the overfitting/underfitting trade-off. However computing the scores on the training set can be computationally expensive and is not strictly required to select the parameters that yield the best generalization performance. |

### 3.3.3 *Model Results*

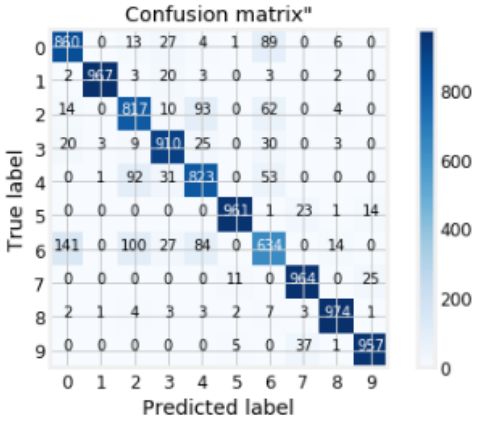
| Fit Score | % Accuracy | Fit Time | Score Time | Predict Time | Precision | Recal | F1-Score |
|---|---|---|---|---|---|---|---|
| 0.8961 | **89%** | 25462.3460 | 0.0 | 2.6526 | 0.89 | 0.89 | 0.89 |

**Classification Report:**

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Class0 | 0.83 | 0.86 | 0.84 | 1000 |
| Class1 | 0.99 | 0.97 | 0.98 | 1000 |
| Class2 | 0.79 | 0.82 | 0.80 | 1000 |
| Class3 | 0.89 | 0.91 | 0.90 | 1000 |
| Class4 | 0.80 | 0.82 | 0.81 | 1000 |
| Class5 | 0.98 | 0.96 | 0.97 | 1000 |
| Class6 | 0.72 | 0.63 | 0.67 | 1000 |
| Class7 | 0.94 | 0.96 | 0.95 | 1000 |
| Class8 | 0.97 | 0.97 | 0.97 | 1000 |
| Class9 | 0.96 | 0.96 | 0.96 | 1000 |
| accuracy | | | 0.89 | 10000 |
| macro avg | 0.89 | 0.89 | 0.89 | 10000 |
| weighted avg | 0.89 | 0.89 | 0.89 | 10000 |

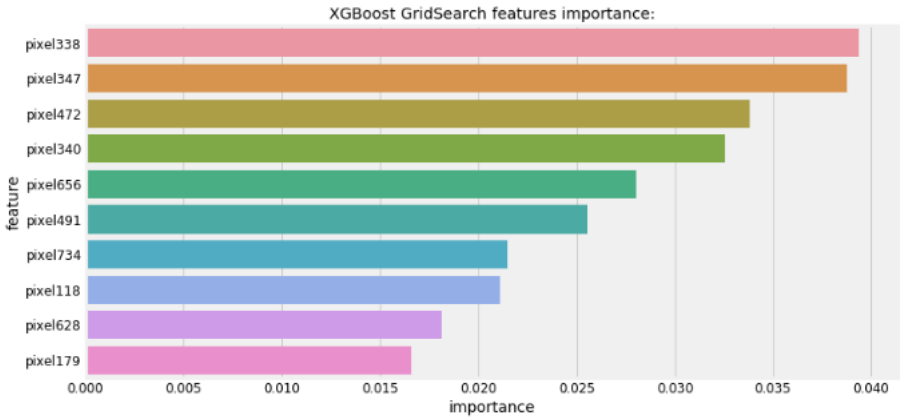
**Top 10**

**Confusion Matrix Report:**



Lab 3 (week 9)

## Feature Importance:



# 4 Neural Networks with Keras

Python package: keras
Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation.

- Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).
- Supports both convolutional networks and recurrent networks, as well as combinations of the two.
- Runs seamlessly on CPU and GPU.

## 4.1 Analysis and Models

### 4.1.1 *Data Preprocessing*

Data preprocessing for this model is described above in section 1.3 and 1.4.

Additionally, one hot encoding was performed to convert class vectores to binary class matrices. This means that a column will be created for each output category and a binary variable is inputted for each category.

## 4.2 Model Baseline - Single Layer Neural Network

### 4.2.1 *Model Details*

This is a Keras Sequential model, with two Dense layers. The Sequential model is a linear stack of layers. The input_shape for this model is the number of pixels for this dataset of images which is 784.

The compile method, which configures the model for training, uses an 'adam' optimizer and for loss the 'categorical_crossentropy' objective function.

This models most notable training (fit) parameters are:

- epochs=300
- batch_size=1000
- verbose=2
- shuffle=True

#### 4.2.1.1 Parameters - Input Dense Layer

```
{'batch_size': 1000,
 'epochs': 300,
 'steps': None,
 'samples': 48000,
 'verbose': 2,
 'do_validation': True,
 'metrics': ['loss', 'acc', 'val_loss', 'val_acc']}
```

| Parameter and Value | Description |
|---|---|
| **units=784** | Positive integer, dimensionality of the output space |
| **input_dim=784** | Input shape, nD tensor |
| **kernel_initializer='normal'** | Initializer for the kernel weights matrix |
| **activation='relu'** | Activation function to use. If you don't specify anything, no activation is applied (i.e., 'linear' activation) |

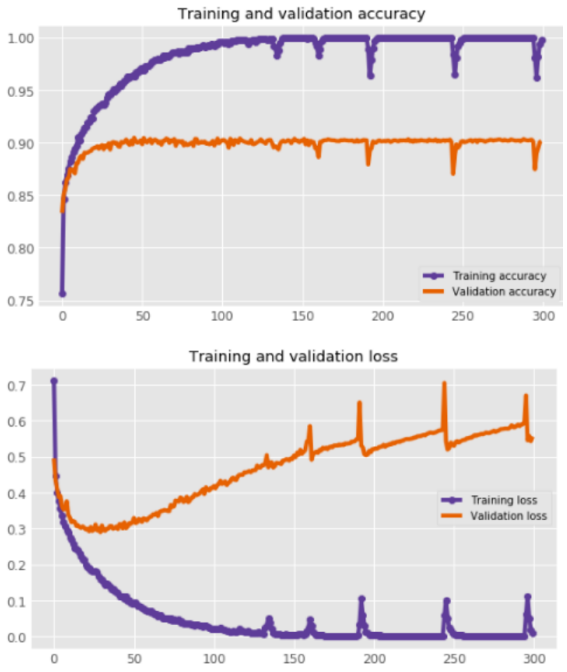
#### 4.2.1.2 Parameters - Output Dense Layer

| Parameter and Value | Description |
|---|---|
| **unit=10** | Positive integer, dimensionality of the output space |
| **kernel_initializer='normal'** | Initializer for the kernel weights matrix |
| **activation='softmax'** | Activation function to use. |

#### 4.2.1.3 Model Summary

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 784)               615440
_________________________________________________________________
dense_2 (Dense)              (None, 10)                7850
=================================================================
Total params: 623,290
Trainable params: 623,290
Non-trainable params: 0
```

**Training and Validation Accuracy and Validation Loss**



Lab 3 (week 9)

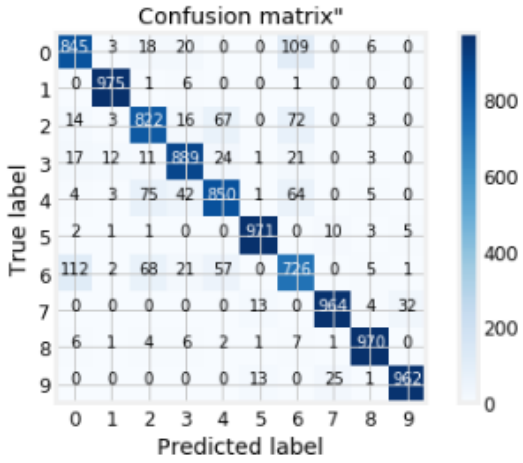
### 4.2.2 *Model Results*

| Fit Score | % Accuracy | Fit Time | Score Time | Predict Time | Precision | Recal | F1-Score |
|---|---|---|---|---|---|---|---|
| 0.897 | **90%** | 676.8362 | 0.6127 | 0.0 | 0.90 | 0.90 | 0.90 |

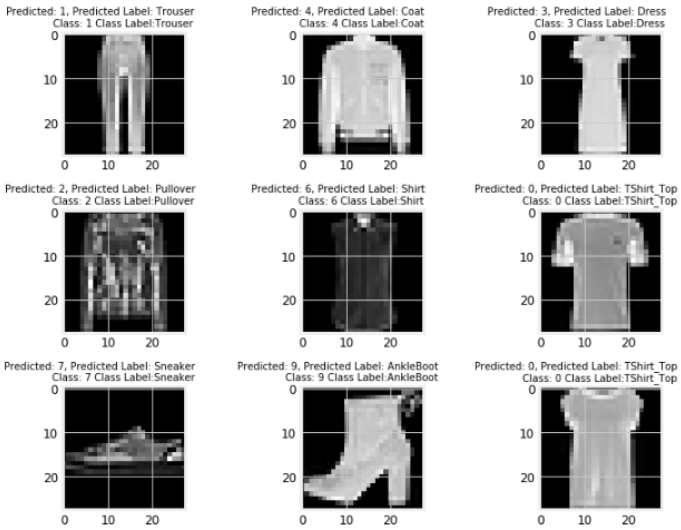
**Classification Report:**

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Class0 | 0.84 | 0.84 | 0.84 | 1000 |
| Class1 | 0.99 | 0.97 | 0.98 | 1000 |
| Class2 | 0.82 | 0.82 | 0.82 | 1000 |
| Class3 | 0.91 | 0.89 | 0.90 | 1000 |
| Class4 | 0.81 | 0.85 | 0.83 | 1000 |
| Class5 | 0.98 | 0.97 | 0.97 | 1000 |
| Class6 | 0.73 | 0.73 | 0.73 | 1000 |
| Class7 | 0.95 | 0.96 | 0.96 | 1000 |
| Class8 | 0.97 | 0.97 | 0.97 | 1000 |
| Class9 | 0.96 | 0.96 | 0.96 | 1000 |
| accuracy | | | 0.90 | 10000 |
| macro avg | 0.90 | 0.90 | 0.90 | 10000 |
| weighted avg | 0.90 | 0.90 | 0.90 | 10000 |

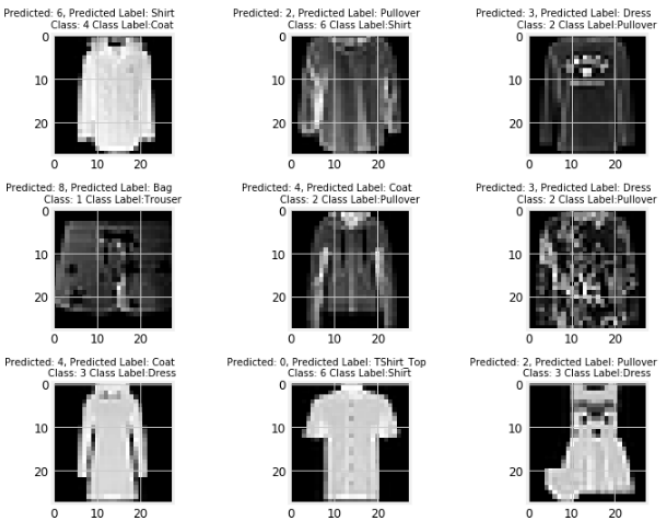
**Confusion Matrix Report:**




**Sample Correct Predicted Images:**




**Sample Incorrect Predicted Images:**



## 4.3 Model - Convelutional Neural Network

Deciding on 3x3 as the optimal kernel size for CNN, see here for details.

### 4.3.1 *Model Details*

This is a Keras Sequential model, with three 2D convelutional layers. The Sequential model is a linear stack of layers. The input_shape for this model is a 3x3x1. The 2D convolution filter like 3x3 will always have a third dimension in size. The third dimension is equal to the number of channels of the input image. For gray-scale images such as this dataset, that has 1 black and white channel, the third dimension is a 1. Our input_shape is then, (28, 28, 1)

The compile method, which configures the model for training, uses an 'adam' optimizer and for loss the 'categorical_crossentropy' objective function.

This models most notable training (fit) parameters are:

- epochs=300
- batch_size=1000
- verbose=2
- shuffle=True

#### 4.3.1.1 Parameters - Layer 1: Conv2D

| Parameter and Value | Description |
|---|---|
| **filters=28** | Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution) |
| **kernel_size=3** | An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions. |
| **padding='same'** | One of 'valid' or 'same' |
| **activation='relu'** | Activation function to use. |
| **kernel_initializer='he_normal'** | Initializer for the kernel weights matrix |
| **input_shape=(28, 28, 1)** | When using this layer as the first layer in a model, this keyword argument is required. Tuble of integers specifing the image shape. |

#### 4.3.1.2 Parameters - Layer 2: MaxPooling2D

| Parameter and Value | Description |
|---|---|
| **pool_size=(2,2)** | Integer or tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2,2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimesions. |

#### 4.3.1.3 Parameters - Layer 3: Dropout

| Parameter and Value | Description |
|---|---|
| **rate=0.25** | Float between 0 and 1. Fraction of the input units to drop. |

#### 4.3.1.4 Parameters - Layer 4: Conv2D

| Parameter and Value | Description |
|---|---|
| **filters=64** | Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution) |
| **kernel_size=3** | An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions. |
| **padding='same'** | One of 'valid' or 'same' |
| **activation='relu'** | Activation function to use. |

#### 4.3.1.5 Parameters - Layer 5: MaxPooling2D

| Parameter and Value | Description |
|---|---|
| **pool_size=(2,2)** | Integer or tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2,2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimesions. |

#### 4.3.1.6 Parameters - Layer 6: Dropout

| Parameter and Value | Description |
|---|---|
| **rate=0.25** | Float between 0 and 1. Fraction of the input units to drop. |

#### 4.3.1.7 Parameters - Layer 7: Conv2D

| Parameter and Value | Description |
|---|---|
| **filters=128** | Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution) |
| **kernel_size=3** | An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions. |
| **activation='relu'** | Activation function to use. |

#### 4.3.1.8 Parameters - Layer 8: Dropout

| Parameter and Value | Description |
|---|---|

| | |
|---|---|
| **rate=0.4** | Float between 0 and 1. Fraction of the input units to drop. |

#### 4.3.1.9 Parameters - Layer 9: Flatten

| Parameter and Value | Description |
|---|---|
| | |

#### 4.3.1.10 Parameters - Layer 10: Dense

| Parameter and Value | Description |
|---|---|
| **unit=748** | Positive integer, dimensionality of the output space |
| **activation='relu'** | Activation function to use. |

#### 4.3.1.11 Parameters - Layer 11: Dropout

| Parameter and Value | Description |
|---|---|
| **rate=0.3** | Float between 0 and 1. Fraction of the input units to drop. |

#### 4.3.1.12 Parameters - Layer 12: Dense

| Parameter and Value | Description |
|---|---|
| **unit=10** | Positive integer, dimensionality of the output space |
| **activation='softmax'** | Activation function to use. |

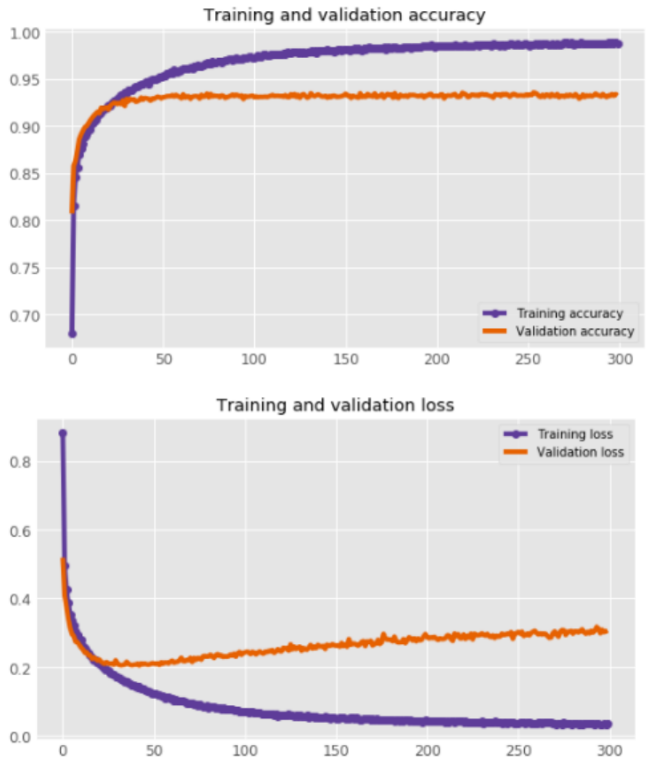
Lab 3 (week 9)

### 4.3.1.13 Model Summary

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d_4 (Conv2D) | (None, 28, 28, 28) | 280 |
| max_pooling2d_3 (MaxPooling2 | (None, 14, 14, 28) | 0 |
| dropout_5 (Dropout) | (None, 14, 14, 28) | 0 |
| conv2d_5 (Conv2D) | (None, 12, 12, 64) | 16192 |
| max_pooling2d_4 (MaxPooling2 | (None, 6, 6, 64) | 0 |
| dropout_6 (Dropout) | (None, 6, 6, 64) | 0 |
| conv2d_6 (Conv2D) | (None, 4, 4, 128) | 73856 |
| dropout_7 (Dropout) | (None, 4, 4, 128) | 0 |
| flatten_2 (Flatten) | (None, 2048) | 0 |
| dense_7 (Dense) | (None, 784) | 1606416 |
| dropout_8 (Dropout) | (None, 784) | 0 |
| dense_8 (Dense) | (None, 10) | 7850 |

Total params: 1,704,594
Trainable params: 1,704,594
Non-trainable params: 0

**Training and Validation Accuracy and Validation Loss**




### 4.3.2 *Model Results*

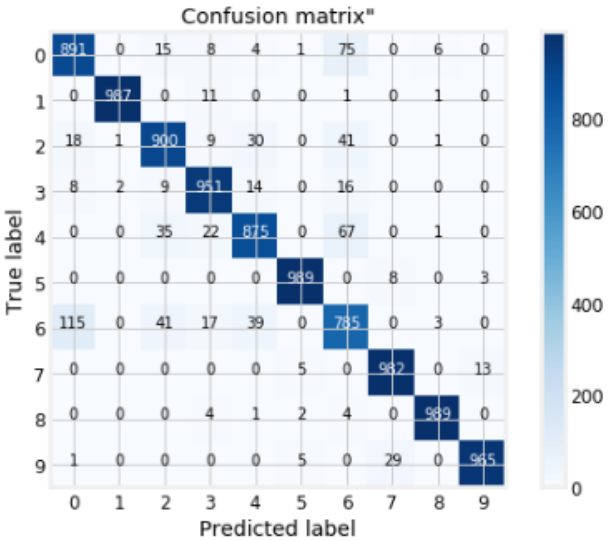
| Fit Score | % Accuracy | Fit Time | Score Time | Predict Time | Precision | Recal | F1-Score |
|---|---|---|---|---|---|---|---|
| 0.9314 | 93% | 24986.7392 | 4.5842 | 0.0 | 0.93 | 0.93 | 0.93 |

**Classification Report:**

Each of the classes highlighted in green below represent significant improvements in predicting these classes correctly over all other models evaluated in this report.

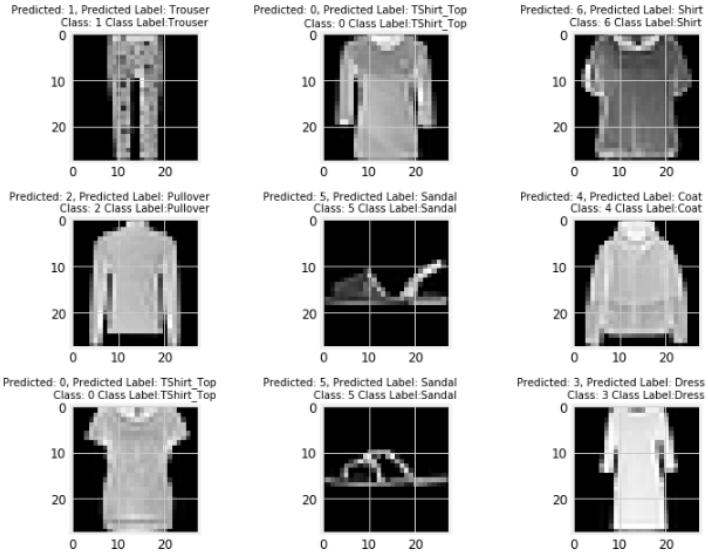
| | precision | recall | f1-score | support |
|---|---|---|---|---|
| Class0 | 0.86 | 0.89 | 0.88 | 1000 |
| Class1 | 1.00 | 0.99 | 0.99 | 1000 |
| Class2 | 0.90 | 0.90 | 0.90 | 1000 |
| Class3 | 0.93 | 0.95 | 0.94 | 1000 |
| Class4 | 0.91 | 0.88 | 0.89 | 1000 |
| Class5 | 0.99 | 0.99 | 0.99 | 1000 |
| Class6 | 0.79 | 0.79 | 0.79 | 1000 |
| Class7 | 0.96 | 0.98 | 0.97 | 1000 |
| Class8 | 0.99 | 0.99 | 0.99 | 1000 |
| Class9 | 0.98 | 0.96 | 0.97 | 1000 |
| accuracy | | | 0.93 | 10000 |
| macro avg | 0.93 | 0.93 | 0.93 | 10000 |
| weighted avg | 0.93 | 0.93 | 0.93 | 10000 |

**Confusion Matrix Report:**

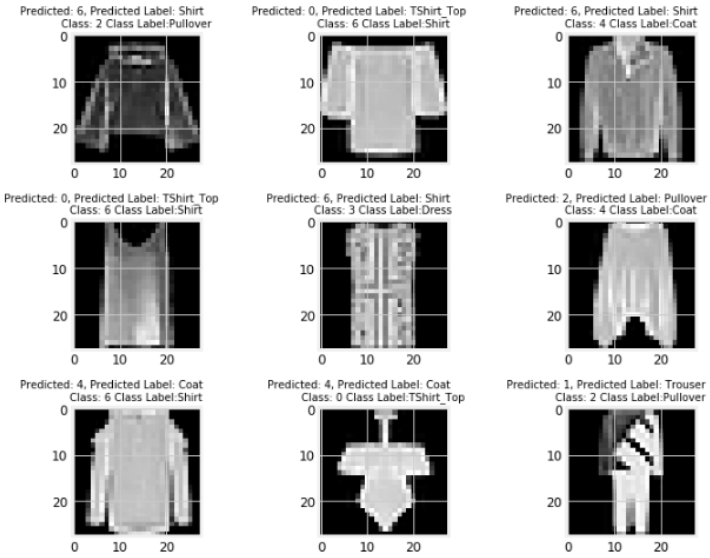


Lab 3 (week 9)

## Sample Correct Predicted Images:




## Sample Incorrect Predicted Images:

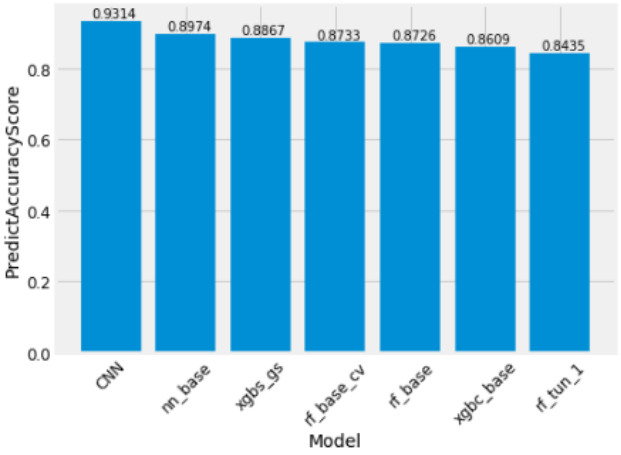


# 5 Algorithm Performance Comparison

*Model Performance Ranking Bar Graph, sorted by PredictAccuracyScore descending:*

Overall, the Convelutional Neural Network had the best predictor accuracy score, however it's compute time was second to last.

The CNN performed significantly better than all other models in classifing the more challenging images (Class6-Shirt, Class4-Coat, Class2-Pullover) . Below is a breakdown of each models f1-score per class prediction.




| | Model | Class0 | Class1 | Class2 | Class3 | Class4 | Class5 | Class6 | Class7 | Class8 | Class9 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | CNN | 0.88 | 0.99 | 0.90 | 0.94 | 0.89 | 0.99 | 0.79 | 0.97 | 0.99 | 0.97 |
| 1 | nn_base | 0.84 | 0.98 | 0.82 | 0.90 | 0.83 | 0.97 | 0.73 | 0.96 | 0.97 | 0.96 |
| 2 | xgbs_gs | 0.84 | 0.98 | 0.80 | 0.90 | 0.81 | 0.97 | 0.67 | 0.95 | 0.97 | 0.96 |
| 3 | xgbs_base | 0.82 | 0.97 | 0.76 | 0.86 | 0.77 | 0.96 | 0.61 | 0.93 | 0.96 | 0.94 |
| 4 | rf_tun_1 | 0.82 | 0.98 | 0.74 | 0.86 | 0.75 | 0.95 | 0.60 | 0.91 | 0.96 | 0.93 |
| 5 | rf_base | 0.83 | 0.97 | 0.78 | 0.88 | 0.78 | 0.97 | 0.64 | 0.94 | 0.96 | 0.95 |

*Model Performance Ranking Bar Graph, sorted by log of Total Compute Time ascending:*

| | ModelName | TestAccuracyScore | PredictAccuracyScore | TrainTime | TestTime | ScoreTime | PredictTime | TotalTime |
|---|---|---|---|---|---|---|---|---|
| 1 | CNN | 0.931400 | 0.9314 | 24986.739215 | 0.000000 | 4.584205 | 0.000000 | 24991.323420 |
| 0 | nn_base | 0.897400 | 0.8974 | 676.836291 | 0.000000 | 0.612764 | 0.000000 | 677.449054 |
| 4 | xgbs_gs | 0.896167 | 0.8867 | 25462.346012 | 1.900499 | 0.000000 | 2.652688 | 25466.899199 |
| 1 | rf_base_cv | 0.883280 | 0.8733 | 59.931356 | 0.000000 | 0.987812 | 0.520665 | 61.439834 |
| 0 | rf_base | 0.880667 | 0.8726 | 84.922047 | 0.641522 | 0.000000 | 0.533718 | 86.097286 |
| 3 | xgbc_base | 0.868833 | 0.8609 | 654.543446 | 0.609808 | 0.000000 | 0.761276 | 655.914530 |
| 2 | rf_tun_1 | 0.851000 | 0.8435 | 25.983807 | 0.433044 | 0.000000 | 0.446281 | 26.863133 |

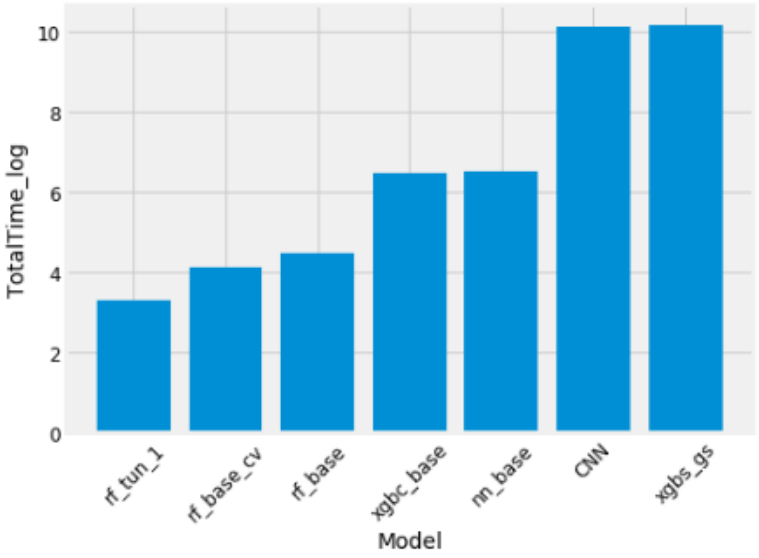
Lab 3 (week 9)

## *Model Performance Ranking Table, sorted by Total Compute Time ascending:*

***Note - The compute time for these models are based on the processing power of the system. These were ran locally, on an Intel Core i7-8650U CPU@1.90 GHz (8 CPUs)*

The tuned Random Forest had the fastest compute times, however it was last in prediction accuracy scoring at 84%.

The tradeoff between these models is accuracy versus compute time. It would depend on the use case in chosing which model to select.




| | ModelName | TestAccuracyScore | PredictAccuracyScore | TrainTime | TestTime | ScoreTime | PredictTime | TotalTime |
|---|---|---|---|---|---|---|---|---|
| 2 | rf_tun_1 | 0.851000 | 0.8435 | 25.983807 | 0.433044 | 0.000000 | 0.446281 | 26.863133 |
| 1 | rf_base_cv | 0.883280 | 0.8733 | 59.931356 | 0.000000 | 0.987812 | 0.520665 | 61.439834 |
| 0 | rf_base | 0.880667 | 0.8726 | 84.922047 | 0.641522 | 0.000000 | 0.533718 | 86.097286 |
| 3 | xgbc_base | 0.868833 | 0.8609 | 654.543446 | 0.609808 | 0.000000 | 0.761276 | 655.914530 |
| 0 | nn_base | 0.897400 | 0.8974 | 676.836291 | 0.000000 | 0.612764 | 0.000000 | 677.449054 |
| 1 | CNN | 0.931400 | 0.9314 | 24986.739215 | 0.000000 | 4.584205 | 0.000000 | 24991.323420 |
| 4 | xgbs_gs | 0.896167 | 0.8867 | 25462.346012 | 1.900499 | 0.000000 | 2.652688 | 25466.899199 |

# 6 Appendix - Submission Items

- Report with graphics
- Supporting notebook for the report
- Description of your tool and methodology if you used another statistical/software package

# 7 Appendix B - Behind Fashion-MNIST

The original MNIST dataset contains a lot of handwritten digits. Members of the AI/ML/Data Science community love this dataset and use it as a benchmark to validate their algorithms. In fact, MNIST is often the first dataset researchers try. "*If it doesn't work on MNIST, it **won't work** at all*", they said. "*Well, if it does work on MNIST, it may still fail on others.*"

- **MNIST is too easy.** Convolutional nets can achieve 99.7% on MNIST. Classic machine learning algorithms can also achieve 97% easily. Check out our side-by-side benchmark for Fashion-MNIST vs. MNIST, and read "Most pairs of MNIST digits can be distinguished pretty well by just one pixel."
- **MNIST is overused.** In this April 2017 Twitter thread, Google Brain research scientist and deep learning expert Ian Goodfellow calls for people to move away from MNIST.
- **MNIST can not represent modern CV tasks**, as noted in this April 2017 Twitter thread, deep learning expert/Keras author François Chollet.