INSY-5378-001: Data Science - A Programming Approach
Christian Kamarga
University of Texas Arlington

Individual Project - Spring 2022

You need to work on a popular Fashion MNIST dataset for this project. The dataset includes tiny images of fashion pieces. The objective is to create a set of supervised learning models that can predict the type of item based on its image. You can use all different models that you learned about them in this course for yourr work. Keep in mind that this is a project, not a class assignmnet. So, not all steps are predetermined and you have more flexibility, and the final outcome is likely to be more detailed.

In order to load the dataset you need to have `tensorflow V2` on your computer. Use the following code to install the package

In [1]:
```
# !pip install --upgrade tensorflow
```

You can also check the version of it using the following code.

In [2]:
```
import tensorflow as tf
tf.__version__
```

Out[2]:
```
'2.8.0'
```

Now, it's time to load the dataset

In [3]:
```
from tensorflow import keras
fashion_mnist = keras.datasets.fashion_mnist
(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()
```

As can be seen from the above code, the dataset was divided into train and test sets. Let's take a look at the X_train

In [4]:
```
X_train.shape
```

Out[4]:
```
(60000, 28, 28)
```

As it is clear, the train dataset (`X_train`) contains 60,000 images of size 28 x 28. We can visualize one of the images using the following code:

In [5]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
%matplotlib inline

sample_image = X_train[10]
plt.imshow(sample_image, cmap='binary')
plt.axis('off')
plt.show()
```

The `y_train` also includes values between 0 and 9. Each represents a particular category. For example, we can check the value of `y_train` for the above image.

In [6]:
```
y_train[10]
```

Out[6]:
```
0
```

The above code shows that the image belongs to category 0. To get the associated label with each category, you can use the following code:

In [7]:
```
class_names = ['T-shirt/top','Trouser','Pullover','Dress','Coat','Sandal','Shirt','Sneaker','Bag','Ankle boot']
print(class_names[y_train[10]])
```

```
T-shirt/top
```

Now, it's your turn,

- **Task1:** Use the train set to train various supervised models and evaluate their performance using the test set.
  - Use different supervised learning models.
  - Use different metrics such as **accutacy**, **precision**, **AUC**, and ... in your model evaluation.
  - It is not enough to report the metrics. It is crucial that you interpret the metrics for each model and compare them across different models.
  - You may need to use the cross validation methods for hyperparameter selection.
  - Specify the model that outperforms the other models.
- **Task2:** Use the best model to predict your own fashion pieces.
  - Take a picture of five fashion pieces of your own (take pictures in square format).
  - Resize images to the correct size (28,28).
  - Grayscale your images.
  - Visualize all the images side by side
  - Use the best model in Task 1 to predict the label of each of your own images.
  - How accurate is the final result?

First things first, import some basic necessities for my models throughout the report.

In [8]:
```
# Basic imports that will be needed throughout the report.
import pandas as pd
import numpy as np

# The model imports. Listed in top-down chronological order for this report (i.e., Logistic Regression is first
# is second, and so forth).
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier, export_graphviz
from sklearn.ensemble import RandomForestClassifier

from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, roc_auc_score
import seaborn as sns; sns.set()
from sklearn.model_selection import train_test_split

# Image import will be needed for Task 2.
from PIL import Image
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
```

Task 1:

We are going to use the training set to train various Supervised Learning models. Our goal is to determine which algorithm will be best to implement Task 2 where we will accept new data. Each model will be evaluated through different performance metrics:

- **Wall-Clock time**, or just Wall time. This measure will be analyzed at the fitting for every model throughout this report. It report on how long it takes for the computer program to fit the model. Assuming that time is of the essence for our fashion dataset, obtaining efficient and high performing models within little time is ideal.
- **Accuracy** will be measured for in-sample (training) and out-of-sample (testing) data for each of the models. Accuracy reports on how accurately the model predicts, and compares it to the in-sample output (**y_train**). An ideal and well-performing model would have a better in-sample accuracy than an out-of-sample accuracy. If otherwise, that would indicate that something is truly off in the model.
- **Precision** is a ratio of the true positives to the sum of true and false positives. In other words, it analyzes how many of the samples of the dataset are classified as positives.
- **Recall** is a ratio of the true positives to the sum of the true positives and the false negatives. In other words, it reports how many of the samples of the dataset are actually positives.

Note: Precision and Recall can be considered important for the machine learning process, but in many business contexts, one can be more important than the other. For this context with the fashion dataset, it ***depends*** </i>. *If we are a clothing business that wants to create an automated identifier in registering new products to the online website, we may prioritize reporting Precision over Recall in that we want clothes to be classified accordingly to their categories.*

*Since the goal of this context is to classify images of clothing to their respective categories, only models for Classification will be used. Regression models, such as Linear Regression and Polynomial Regression, are not suitable for this context because our target variable (**y_train**) is not continuous.*

*Training the models will require parameters with 2 dimensions. In the cell below, we will reshape **X_train** and **X_test**. This is flattening the images stored in the arrays.*

In [9]:
```
X_train_arr = X_train.reshape(60000, 28 * 28) # 60000 * 784;
X_test_arr = X_test.reshape(10000, 28 * 28) # 10000 * 784;

print("X_train.shape: ", X_train.shape) # (60000, 28, 28)
print("X_train_arr.shape: ", X_train.shape) # (60000, 784)
print("X_test.shape: ", X_test.shape) # (10000, 28, 28)
print("y_train.shape: ", y_train.shape) # (60000, )
print("y_test.shape: ", y_test.shape) # (10000, )
```

```
X_train.shape:  (60000, 28, 28)
X_train_arr.shape:  (60000, 28, 28)
X_test.shape:  (10000, 28, 28)
y_train.shape:  (60000,)
y_test.shape:  (10000,)
```

1. Logistic Regression

1a. Description

Logistic Regression is an algorithm that models the relationship between a binary dependent variable and one or more features (independent variables).

In [10]:
```
# from sklearn.linear_model import LogisticRegression
logistic_regression_model = LogisticRegression(max_iter = 1000)

# With X_train reshaped into a 2D matrix named X_train_arr, this line will executre to fit the logistic regress
%time logistic_regression_model.fit(X_train_arr, y_train)
```

```
Wall time: 1min 26s
C:\Users\chris\anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:763: ConvergenceWarning: lbfgs fai
led to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

Out[10]:
```
LogisticRegression(max_iter=1000)
```

Here, it can be observed it does not take too long to fit the Logistic Regression model. Logistic Regression is a fairly simple and straightforward algorithm to implement and train. Let's observe how it performs as predictive models:

1b. Accuracy

In [11]:
```
# By predicting the model on training data, the accuracy for in-sample data can be obtained.
y_train_hat = logistic_regression_model.predict(X_train_arr)
y_test_hat = logistic_regression_model.predict(X_test_arr)

# Displays the Accuracy scores for in-sample (training) and out-of-sample (testing) data
print("In-sample Accuracy Score: ", accuracy_score(y_train, y_train_hat, normalize = True) * 100) # 87.9%
print("Out-of-sample Accuracy Score: ", accuracy_score(y_test, y_test_hat, normalize = True) * 100) # 83.2%
```

```
In-sample Accuracy Score:  87.92333333333333
Out-of-sample Accuracy Score:  83.22
```

*We can see here that this Logistic Regression model performs with in-sample (training) data better than with out-of-sample (testing) data. This is quite ideal for our models. Still, we need to observe how well it is determining the ten categories from **class_names**.*

1c. Classification Report

In [12]:
```
# By setting target_names = class_names, the classification report will generate the performance metrics with t
# clothing categories for easier comprehension.
print(classification_report(y_test, y_test_hat, target_names = class_names))
```

	precision	recall	f1-score	support
T-shirt/top	0.80	0.80	0.80	1000
Trouser	0.95	0.95	0.95	1000
Pullover	0.72	0.72	0.72	1000
Dress	0.82	0.84	0.83	1000
Coat	0.71	0.77	0.74	1000
Sandal	0.93	0.89	0.91	1000
Shirt	0.63	0.55	0.59	1000
Sneaker	0.91	0.93	0.92	1000
Bag	0.91	0.92	0.92	1000
Ankle boot	0.93	0.94	0.93	1000
accuracy			0.83	10000
macro avg	0.83	0.83	0.83	10000
weighted avg	0.83	0.83	0.83	10000

A Classification Report includes the metrics for Precision, Recall, and Accuracy. We have already seen the Accuracy for this model in the above cell. In this Confusion Matrix, we can see the quantities for each of the class labels.

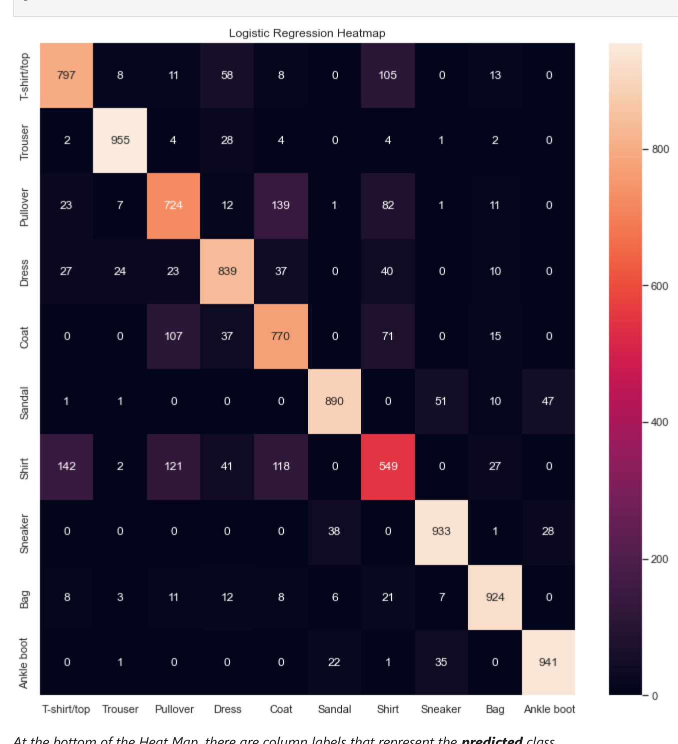
As a reminder: *The **Precision** is a ratio that displays how much of the data is classified as a positive. The **Recall** is a ratio that indicates how much of the data is actually classified correctly.*

Each of the clothing categories seems to be performing at least fairly well. The Precision and Recall metrics are closely high. Since the model was reported with proficient accuracy at least 80% above, this is not really a surprise.

Let's observe how sample quantities of the dataset are being classified in a Confusion Matrix.

1d. Confusion Matrix

In [13]:
```
# This variable takes the output for out-of-sample data and the predicted model's predicitons.
cf_matrix = confusion_matrix(y_test, y_test_hat)

# Storing the confusion matrix in a dataframe with the class names labelled for the rows and columns.
cm_df = pd.DataFrame(cf_matrix,
                     index = class_names,
                     columns = class_names)

# Sets the size for the Heatmap.
fig, ax = plt.subplots(1, figsize = (12, 12))

# fmt = 'g' means the heatmap will display the values in each of the cells like in the above confusion matrix.
# Without it, it would display in scientific notation (i.e. 586 = 5.9e+02).
sns.heatmap(cm_df, annot = True, fmt = 'g')
plt.title("Logistic Regression Heatmap")
plt.show()
```



*At the bottom of the Heat Map, there are column labels that represent the **predicted** class.*

*At the left of the Heat Map, there are row labels that represent the **actual** class.*

The Heat Map provides an aesthetically appealing visual in understanding how quantities of the samples of the fashion dataset are being classified by our Logistic Regression model. The dark areas have low quantities, while the brighter colored areas contain higher quantities. In this heat map, seeing the bright colored cells in a diagonal formation indicates majority of the dataset is being classified accurately, or as true positives. This is also not too surprising, considering that the accuracy scores for this Logistic Regression model was reported above 80%.

1e. Interpretation

It's still too early to determine whether this Logistic Regression model is the top-performing model for our Task 2. *However, the Logistic Regression model's simplicity and linear learning capabilities make it a candidate. For now, it will be considered as this report goes on. Let's move on to the next model.*

2. K-Nearest Neighbors (K-NN)

2a. Description

This K-Nearest Neighbors algorithm is another classification algorithm. This model was selected to analyze the data because K-NN is one of the simplest classification algorithms to interpret and efficient to run.

*However, we need to provide a value for one of K-NN's parameters (K), the neighborhood cardinality. By providing a value of K, the algorithm is searching for the K-closest data points in the dataset. Selecting the optimal value for K is the fulcrum for the quality of the K=NN predictive models, which is why we are running **Grid Search with Cross Validation**.*

Grid Search with Cross-Validation works by testing a subset of different values for the K parameter in the K-NN model.

Naturally with any method involving Cross Validation, the dataset is being split into K folds. Splitting the dataset cannot be done without further consideration. It could lead to the split datasets being imbalanced and biased. Thankfully, we can utilize the Stratified Cross Validation. This will split the dataset among each individual fold so that the proportion of different classes/categories are identical in fair.

*Below, we will be running **Stratified Cross Validation with Grid Search**.*

In [14]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV, StratifiedKFold

# This variable stores the K-NN model.
KNN_model = KNeighborsClassifier()

# It is going to experiment with siz different parameter values of K. [1 through 6].
param_grid = {'n_neighbors': [1, 2, 3, 4, 5, 6]}

# Stratified 5-folds. That means for each fold, 4 of them are for training and 1 of them is for testing.
cv = StratifiedKFold(n_splits = 5, random_state = 0, shuffle = True)
grid = GridSearchCV(KNN_model, param_grid, cv = cv, scoring='accuracy',
                    return_train_score = True)
%time grid.fit(X_train_arr, y_train)

# This displays the best value for the K hyperparameter of K-NN.
print("Best Parameter: {}".format(grid.best_params_))
# This displays the average of the cross validation 5-folds.
print("Best Cross Validation Score: {}".format(grid.best_score_))
```

```
Wall time: 1h 11min 52s
Best Parameter: {'n_neighbors': 4}
Best Cross Validation Score: 0.8570166666666665
```

2b. Interpretation

No algorithm comes without weaknesses. In the effort of searching for an optimal parameter for the K-NN model, it required an extensively long time to find before fitting the model.

*After all, with the 6 hyperparameters being tested in a stratified 5-fold, that meant it was running about **30** K-NN models. Additionally, the training data involves **60,000** samples of the fashion dataset with around 784 features. For K-NN, it is not computationally simple to determine the feature importance for our context of multiclass classification.*

For the reporting efficiency, the K-NN model will not be considered for our top-performing model. Moving onto the next model.

3. Naive Bayes

3a. Description

*Naive Bayes models offer simple classifications at fast speeds. They are very suitable for high-dimensional datasets. Especially considering that the **X_train_arr** from the fashion dataset contains 60,000 images (each with a size of 28-by-28 pixels), using a Naive Bayes model can be appropriate for this context. There are multiple variations of Naive-Bayes:*

- *Bernoulli-Naive Bayes*
- *Multinomial-Naive Bayes*
- *Gaussian-Naive Bayes*

*Only **Gaussian-Naive Bayes** will be considered. Why? The Gaussian-Naive Bayes works under the assumption that our class labels, **class_names**, as a Gaussian distribution, which is another term for Normal Distribution.*

*The **Bernoulli-Naive Bayes** is relevant to contexts in a Bernoulli Process, where the outcome is Binary. Additionally, all features (independent variables) would have to be independent and Boolean. That is not the case at all for the context of this fashion dataset.*

*The **Multinomial-Naive Bayes** considers things as a Multinomial Distribution. It would have been fitting if our features in the Fashion Dataset represented frequency, or count rates. In this case, our dataset's features represent pixels of many considered images, so Multinomial-Naive Bayes is out.*

In [19]:
```
from sklearn.naive_bayes import GaussianNB

# This stores the Gaussian-Naive Bayes model into this variable.
GaussianNB_model = GaussianNB()
%time GaussianNB_model.fit(X_train_arr, y_train)
```

```
Wall time: 1.34 s
```

Out[19]:
```
GaussianNB()
```

As we can see above, it did not take very long for the Kernel to compute this model. As mentioned previously, Naive Bayes models provide fast and simple classifications.

We still need to check their performance metrics. Let's start with checking the accuracy for both in-sample and out-of-sample data.

3b. Accuracy

In [20]:
```
# For the predictive model, must reshape X_test into an array with 2 dimensions to work.
y_train_hat = GaussianNB_model.predict(X_train_arr)
y_test_hat = GaussianNB_model.predict(X_test_arr)

# Computes the accuracy scores for the training and testing datasets.
in_sample_acc = accuracy_score(y_train, y_train_hat, normalize = True) * 100
out_of_sample_acc = accuracy_score(y_test, y_test_hat, normalize = True) * 100

# Displays the accuracy scores.
print("In-sample Accuracy: ", in_sample_acc)
print("Out-of-sample Accuracy: ", out_of_sample_acc)
```

```
In-sample Accuracy:  58.778333333333336
Out-of-sample Accuracy:  58.56
```

The accuracy of training (in-sample) data is higher than that of the testing (out-of-sample) data. This indicates the model was built sufficiently well. However, accuracy scores that are narrowly greater than 50.0% can be deemed as left to be desired, depending on our business context and goals. Here, it will be considered acceptable that the majority of in and out of sample data are accurately predicted with this model.

Now, let's examine its Precision and Recall scores from a Classification Report.

3c. Classification Report

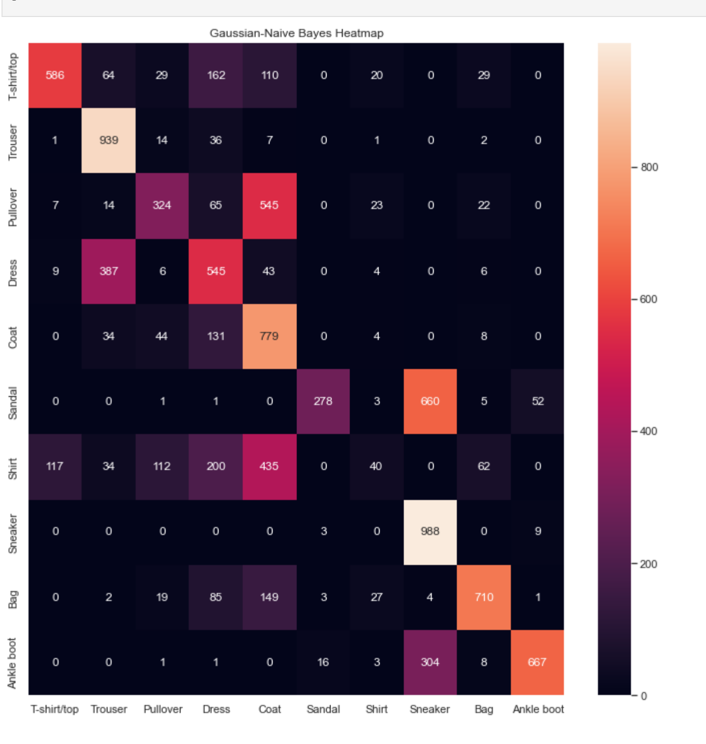
In [21]:
```
# By setting target_names = class_names in the classification_report, it will display the classes as the label
# of the fashion data set. This provides better aesthetics for interpretation.
print(classification_report(y_test, GaussianNB_model.predict(X_test_arr), target_names = class_names))
```

	precision	recall	f1-score	support
T-shirt/top	0.81	0.59	0.68	1000
Trouser	0.64	0.94	0.76	1000
Pullover	0.59	0.32	0.42	1000
Dress	0.44	0.55	0.49	1000
Coat	0.38	0.78	0.51	1000
Sandal	0.93	0.28	0.43	1000
Shirt	0.32	0.04	0.07	1000
Sneaker	0.51	0.99	0.67	1000
Bag	0.83	0.71	0.77	1000
Ankle boot	0.91	0.67	0.77	1000
accuracy			0.59	10000
macro avg	0.64	0.59	0.56	10000
weighted avg	0.64	0.59	0.56	10000

*Precision and Recall for a handful of the class labels are quite low. Shirt only has 32% Precision and 4% Recall rate. Compared to the **Logistic Regression**, this Classification Report for this Gaussian-Naive Bayes shows a lot to be desired. We can anticipate that the Confusion Matrix with the Heatmap in the cell below will be all over the place in terms of dark and bright colored cells.*

3d. Confusion Matrix

In [22]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns; sns.set()

# Stores the Confusion Matrix.
cf_matrix = confusion_matrix(y_test, GaussianNB_model.predict(X_test_arr))

# This dataframe will be used for the Heatmap. Its indices and columns will have the class labels from the fash
cm_df = pd.DataFrame(cf_matrix,
                     index = class_names,
                     columns = class_names)

fig, ax = plt.subplots(1, figsize = (12, 12))

# fmt = 'g' means the heatmap will display the values in each of the cells like in the above confusion matrix.
# Without it, it would display in scientific notation (i.e. 586 = 5.9e+02).
sns.heatmap(cm_df, annot = True, fmt = 'g')
plt.title("Gaussian-Naive Bayes Heatmap")
plt.show()
```



3e. Interpretation

As anticipated, the bright colored cells are scattered in position for this Heatmap. This is comparatively different the consistent elegance from the Heatmap of the Logistic Regression. To reiterate, the lower accuracy scores of the Gaussian-Naive Bayes would indicate higher false positives and false negatives across the classified dataset.

While Naive Bayes is considerably easy to implement, its quality is lacking and may not be selected as the top performing model for Task 2. Moving onto the next model.

4. Decision Tree

4a. Description

Decision Tree classifiers can be suitable for handling non-linear datasets, especially for this case with the provided fashion dataset. It takes samples of the dataset, considers how to divide them based on certain values for the features before splitting off into branches.

*This can incur one of the weaknesses for Decision Tree models: **Overfitting**. A condition where a model performs exceptionally well with in-sample data, but does not handle or accept new data (another term for out-of-sample) data very well. Thankfully, Decision Tree models have a **max_depth** property that limits (or in other words, **preprunes**) how long a branch from the tree can grow. However, this parameter requires selecting an optimal value, which means it's time to use **Grid Search with Cross Validation** again like with the **previous K-NN model in 2a**.*

In [23]:
```
from sklearn.preprocessing import scale

# The Decision Tree model is stored in this variable.
DecisionTree_model = DecisionTreeClassifier()

# Grid Search with Cross Validation is performed to find the optimal value for max_depth.
param_grid = {'max_depth': [12, 13, 14, 15]}

# Stratified 5-fold.
cv = StratifiedKFold(n_splits = 5, random_state=0, shuffle=True)
grid = GridSearchCV(DecisionTree_model, param_grid, cv = cv,
                    return_train_score = True, scoring = 'accuracy')
%time grid.fit(X_train_arr, y_train)

# Reports on the best value for max_depth and the average of the cross validation scores from the stratified 5-
print("Best Parameter: {}".format(grid.best_params_))
print("Best Cross Validation Score: {}".format(grid.best_score_))
```

```
Wall time: 14min 59s
Best Parameter: {'max_depth': 12}
Best Cross Validation Score: 0.8167666666666668
```

Interestingly, it did not take too long to find the best parameter value, unlike the K-NN model. It can be inferred that running a Decision Tree with a specified max_depth takes evidently less time than running a Decision Tree without any pre-pruning. This is due to the branches of the Decision Tree are regulated to grow up until a certain length, which saves on computing time and resources.

*One of the benefits of using Grid Search with Cross Validation is that the **grid.best_estimator_** already contains the fitted model with the optimal parameter.*

In [24]:
```
# No need to initialize or fit the model again with the max_depth parameter. Can call on it with the grid.best_
DecisionTree_model = grid.best_estimator_
```

We still need to determine how well this Decision Tree performs in terms of Accuracy.

4b. Accuracy

In [25]:
```
# Stores the predicted models with testing and training samples of the data.
y_train_hat = DecisionTree_model.predict(X_train_arr)
y_test_hat = DecisionTree_model.predict(X_test_arr)

# Computes the accuracy scores for the training and testing datasets.
in_sample_acc = accuracy_score(y_train, y_train_hat, normalize = True) * 100
out_of_sample_acc = accuracy_score(y_test, y_test_hat, normalize = True) * 100

# Displays the accuracy scores.
print("In-sample Accuracy: ", in_sample_acc)
print("Out-of-sample Accuracy: ", out_of_sample_acc)
```

```
In-sample Accuracy:  88.75333333333333
Out-of-sample Accuracy:  80.30000000000001
```

*Comparing this to the Accuracy scores for the Logistic Regression back in **1b.**, these Accuracy scores for the Decision Tree are comparatively lower. However, they are relatively good if we can consider at least 80% as sufficient. They are also quantitatively higher than the Accuracy scores obtained for the Gaussian-Naive Bayes in **3b.***

We can examine how this Decision Tree model is classifying the clothing categories of the fashion dataset with another Classification report.

4c. Classification Report

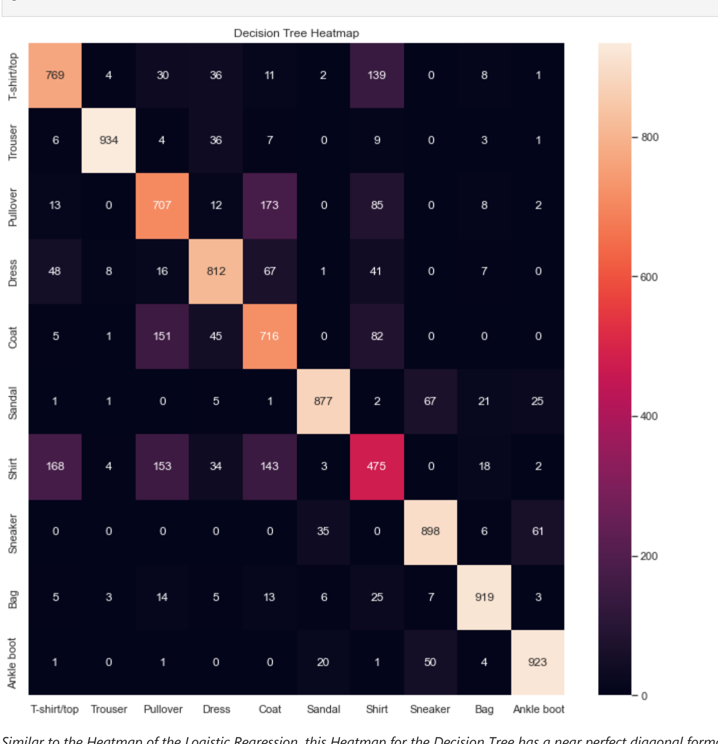
In [26]:
```
print(classification_report(y_test, y_test_hat, target_names = class_names))
```

	precision	recall	f1-score	support
T-shirt/top	0.76	0.77	0.76	1000
Trouser	0.98	0.93	0.96	1000
Pullover	0.66	0.71	0.68	1000
Dress	0.82	0.81	0.82	1000
Coat	0.63	0.72	0.67	1000
Sandal	0.93	0.88	0.90	1000
Shirt	0.55	0.47	0.51	1000
Sneaker	0.88	0.90	0.89	1000
Bag	0.92	0.92	0.92	1000
Ankle boot	0.91	0.92	0.91	1000
accuracy			0.80	10000
macro avg	0.80	0.80	0.80	10000
weighted avg	0.80	0.80	0.80	10000

*Comparing this Classification Report to the Logistic Regression model's in section **1c.**, the Decision Tree model classification qualities are slightly lesser. The Logistic Regression exhibited a higher accuracy and consistently higher overall Precision and Recall rates for the classes. Notably, Shirt in this Confusion Matrix for Decision Trees is smaller than the rest of the labels. It is possible that the images for Shirt inside the MNIST provided dataset are lower quality, and the data set needs to be cleaned.*

4d. Confusion Matrix

In [27]:
```
# Stores the Confusion matrix in this variable.
cf_matrix = confusion_matrix(y_test, y_test_hat)

# Storing the Confusion matrix in a data frame for reporting as a Heatmap below.
cm_df = pd.DataFrame(cf_matrix,
                     index = class_names,
                     columns = class_names)

fig, ax = plt.subplots(1, figsize = (12, 12))

# fmt = 'g' means the heatmap will display the values in each of the cells like in the above confusion matrix.
# Without it, it would display in scientific notation (i.e. 586 = 5.9e+02).
sns.heatmap(cm_df, annot = True, fmt = 'g')
plt.title("Decision Tree Heatmap")
plt.show()
```



Similar to the Heatmap of the Logistic Regression, this Heatmap for the Decision Tree has a near perfect diagonal formation of the brightly colored cells, indicating many of the dataset is classified correctly (true positives). Some cells that are not entirely dark could be outliers, which again suggests there is potential room for data cleaning.

4e. Interpretation

Decision Trees are powerful classifiers, and this situation proves no exception. However, further refinement could be done if it is to compete or surpass the Logistic Regression model. There is one more model that also involves Decision Trees that will be detailed in this report.

5. Random Forest

5a. Description

*Another algorithm that involves Decision Trees. Multiple trees create a forest, which as the name of this algorithm suggests, Random Forest utilizes multiple Decision Trees, depending on the specified value within the **n_estimators** parameter.*

*Given that these are Decision Trees, they will need to be prepruned accordingly like back in **4a.**. Another Grid Search with Cross-Validation will be performed here. However, given that the innate properties of Random Forest involves multiple Decision Trees, it may take considerable time for the below cell to run.*

In [28]:
```
# Initializes the Random Forest model in this variable.
RandomForest_model = RandomForestClassifier() # By default, n_estimators = 100; Still testing for 100 trees.

# Performing Grid Search with Cross Validation to find the optimal value for maximum depth.
param_grid = {'max_depth': [12, 13, 14, 15]}

# Stratified 5-fold cross validation.
cv = StratifiedKFold(n_splits = 5, random_state = 0, shuffle=True)
grid = GridSearchCV(RandomForest_model, param_grid, cv = cv,
                    return_train_score = True, scoring = 'accuracy')
%time grid.fit(X_train_arr, y_train)

# Reports the optimal value for max_depth and the average scores for the stratified 5-fold cross validation.
print("Best Parameter: {}".format(grid.best_params_))
print("Best Cross Validation Score: {}".format(grid.best_score_))
```

```
Wall time: 16min
Best Parameter: {'max_depth': 15}
Best Cross Validation Score: 0.876
```

The reasons it took quite long to finally finish if you consider that:

- *A default/unspecified value of n_estimators is 100. That means 100 Decision Trees are randomly generated in this Random Forest.*
- *Testing 4 hyperparameter values from 12 through 15.*
- *Stratified 5-fold cross validation.*
- *Working with considerably large training data set of 60,000 images.*

However, Decision Trees are powerful tools, and if used and refined right, Random Forest classifiers can be even more powerful. Let us take a look at the accuracy.

5b. Accuracy

In [29]:
```
# No need to initialize or fit the model again with the max_depth parameter. Can call on it with the grid.best_
RandomForest_model = grid.best_estimator_

# Stores the predicted models with testing and training samples of the data.
y_test_hat  = RandomForest_model.predict(X_test_arr)
y_train_hat  = RandomForest_model.predict(X_train_arr)

# Computes the accuracy scores for the training and testing datasets.
in_sample_acc = accuracy_score(y_train, y_train_hat, normalize = True) * 100
out_of_sample_acc = accuracy_score(y_test, y_test_hat, normalize = True) * 100

# Displays the accuracy scores.
print("In-sample Accuracy: ", in_sample_acc)
print("Out-of-sample Accuracy: ", out_of_sample_acc)
```

```
In-sample Accuracy:  96.455
Out-of-sample Accuracy:  86.99
```

*The Accuracy scores for this Random Forest classifier are notably higher overall than those of the **1b. Logistic Regression** and the **(individual) Decision Tree**. In many cases, having more models and more data would improve the accuracy scores overall for a model.*

5c. Classification Report

In [30]:
```
print(classification_report(y_test, y_test_hat, target_names = class_names))
```

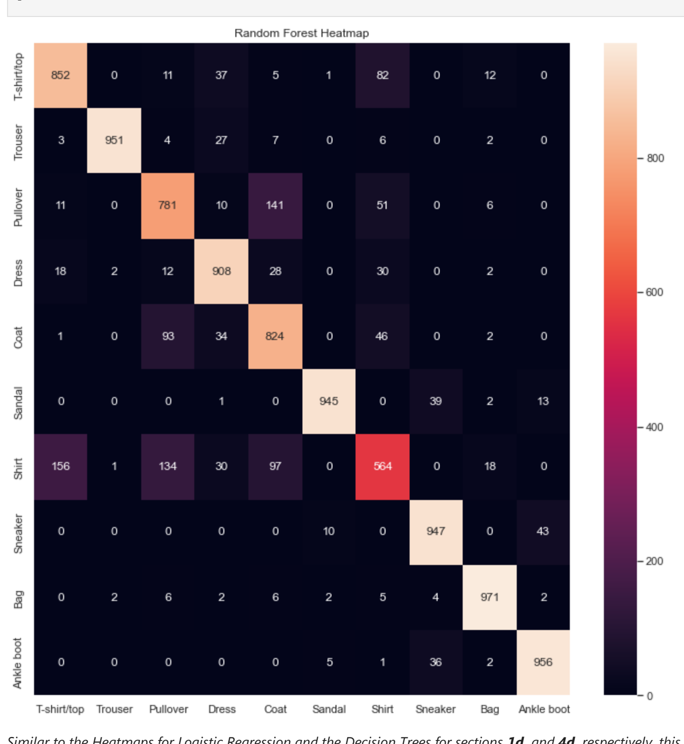
	precision	recall	f1-score	support
T-shirt/top	0.82	0.85	0.83	1000
Trouser	0.99	0.95	0.97	1000
Pullover	0.75	0.78	0.77	1000
Dress	0.87	0.91	0.89	1000
Coat	0.74	0.82	0.78	1000
Sandal	0.98	0.94	0.96	1000
Shirt	0.72	0.56	0.63	1000
Sneaker	0.92	0.95	0.93	1000
Bag	0.95	0.97	0.96	1000
Ankle boot	0.94	0.96	0.95	1000
accuracy			0.87	10000
macro avg	0.87	0.87	0.87	10000
weighted avg	0.87	0.87	0.87	10000

*Overall, the Precision and Recall metrics for each of the class labels are considerably higher with greater consistency, unlike those of the **Gaussian-Naive Bayes** model in **3c.**. Furthermore, this classification report seems to be consistently higher than the Decision Tree's back in section **4c.** However, it seems overall more of the Precision and Recall metrics are roughly around 70% to 80%, which is comparatively lower to those of the Logistic Regression model in section **1c.***

5d. Confusion Matrix

In [31]:

```python
# Stores the Confusion Matrix.
cf_matrix = confusion_matrix(y_test, y_test_hat)

# This dataframe will be used for the Heatmap. Its indices and columns will have the class labels from the fash
cm_df = pd.DataFrame(cf_matrix,
                     index = class_names,
                     columns = class_names)

fig, ax = plt.subplots(1, figsize = (12, 12))

# fmt = 'g' means the heatmap will display the values in each of the cells like in the above confusion matrix.
# Without it, it would display in scientific notation (i.e. 586 = 5.9e+02).
sns.heatmap(cm_df, annot = True, fmt = 'g')
plt.title("Random Forest Heatmap")
plt.show()
```



*Similar to the Heatmaps for Logistic Regression and the Decision Trees for sections **1d.** and **4d.** respectively, this Heatmap sports a diagonal pattern of the positions of the brightly colored cells. Given that the Accuracy Scores for the Random Forest are comparatively higher than the aforementioned models, this comes off as no surprise.*

5e. Interpretation

*Random Forest is a refinement on the existing Decision Tree model. Interestingly, it took less time to run Grid Search with Cross Validation with this model than it did for the Decision Tree model. It is strongly surpasses the Gaussian Naive Bayes model from section **2** across many performance metrics, and in some ways, the Logistic Regression from section **1**.*

It was not an easy pick, but I have ultimately decided to go with Decision Trees. I felt they were relatively simpler to implement than with Random Forests. Although Logistic Regression performed favorably well, I feel it may not be entirely appropriate for this fashion dataset context. The Logistic Regression is a linear learner, and I believe that the features of the images are considerably scattered and "random". If I had to pick a second model, I would pick Random Forests. As stated previously, it is a refined version of the Decision Tree, but it is considerably more complex to implement properly.

Task 2:

Finally, it is time to use the Decision Tree models to predict new and incoming data. I have taken five pictures of my own articles of clothing. For consistency sake, each of my personal images were taken in square format akin to the provided fashion dataset.

Let's visualize them to see what they look like.

In [32]:

```python
from PIL import Image
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
%matplotlib inline

# Defining my images as individual arrays. Each of the images have the shape of (3024, 3024, 3).
task2_image_1 = np.array(mpimg.imread('Task 2/IMG_0956.jpg')) # T-shirt/top
task2_image_2 = np.array(mpimg.imread('Task 2/IMG_0957.jpg')) # Pullover
task2_image_3 = np.array(mpimg.imread('Task 2/IMG_0958.jpg')) # Sneaker
task2_image_4 = np.array(mpimg.imread('Task 2/IMG_0959.jpg')) # Ankle boot
task2_image_5 = np.array(mpimg.imread('Task 2/IMG_0960.jpg')) # Trousers

# Loading the individual arrays into a single collective array. An array of 5 images.
task2_images = np.array([task2_image_1, task2_image_2, task2_image_3, task2_image_4, task2_image_5])

# An array used to index the class_names for the intended labels of the images.
intended = np.array([0, 2, 9, 7, 1])

fig, ax = plt.subplots(1, 5, figsize = (15, 15))
for i, axi in enumerate(ax.flat):
    axi.imshow(task2_images[i])
    axi.set_title(class_names[intended[i]])
    axi.set(xticks=[], yticks=[])

print("Shape for each of the individual images: ", task2_image_1.shape)
print("Shape of task2_images array: ", task2_images.shape)
```

```
Shape for each of the individual images:  (3024, 3024, 3)
Shape of task2_images array:  (5, 3024, 3024, 3)
```

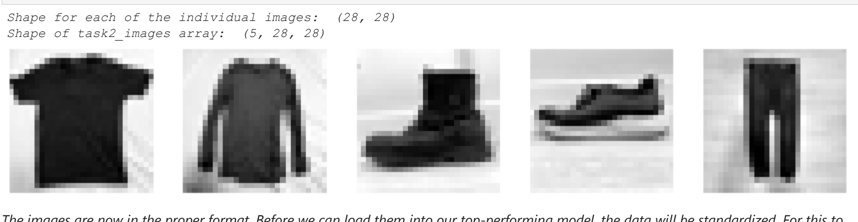
*The collective image array **task2_images** is an array of the 5 image arrays I provided. As we can see here, each of them are colored RGB images of 3024 * 3024 pixels.*

In order to properly observe whether my best performing model can accurately categorize new data, these new images must be converted into a similar format to the images from the provided fashion dataset. Here, every image will be converted to 28 by 28 pixels and grayscaled.

In [33]:

```python
# All of the images are individually opened and resized before the average of their values are computed. The en
# each image is now grayscaled with 28 * 28 pixels.

# Image of a T-shirt/top.
task2_image_1 = Image.open('Task 2/IMG_0956.jpg')
task2_image_1 = task2_image_1.resize((28, 28))
task2_image_1 = np.mean(task2_image_1, -1)

# Image of a pullover.
task2_image_2 = Image.open('Task 2/IMG_0957.jpg')
task2_image_2 = task2_image_2.resize((28, 28))
task2_image_2 = np.mean(task2_image_2, -1)

# Image of an ankle boot.
task2_image_3 = Image.open('Task 2/IMG_0958.jpg')
task2_image_3 = task2_image_3.resize((28, 28))
task2_image_3 = np.mean(task2_image_3, -1)

# Image of a sneaker.
task2_image_4 = Image.open('Task 2/IMG_0959.jpg')
task2_image_4 = task2_image_4.resize((28, 28))
task2_image_4 = np.mean(task2_image_4, -1)

# Image of trousers.
task2_image_5 = Image.open('Task 2/IMG_0960.jpg')
task2_image_5 = task2_image_5.resize((28, 28))
task2_image_5 = np.mean(task2_image_5, -1)

# Loading the individual arrays into a single collective array. An array of 5 images.
task2_images = np.array([task2_image_1, task2_image_2, task2_image_3, task2_image_4, task2_image_5])

fig, ax = plt.subplots(1, 5, figsize=(15, 15))
for i, axi in enumerate(ax.flat):
    axi.imshow(task2_images[i].astype('uint8'), cmap='gray')
    axi.set(xticks=[], yticks=[])

print("Shape for each of the individual images: ", task2_image_1.shape)
print("Shape of task2_images array: ", task2_images.shape)
```

```
Shape for each of the individual images:  (28, 28)
Shape of task2_images array:  (5, 28, 28)
```



The images are now in the proper format. Before we can load them into our top-performing model, the data will be standardized. For this to work properly, the train and test datasets that were initially split at the start of this document will be recombined.

In [34]:

```python
# By vertically stacking the flattened train and test datasets, we obtain the original whole dataset of the fea
X_whole = np.vstack((X_train_arr, X_test_arr)) # 70000 * 784

scaler = StandardScaler()
X_whole = scaler.fit_transform(X_whole)

# With the data scaled, now we can assign them into the proper variables.
X = X_whole[:60000, :] # Previously X_train.
print(y_test.shape)
y_whole = np.hstack((y_train, y_test))
y_whole.shape
```

```
(10000,)
```

Out[34]:

```
(70000,)
```

We can finally load the scaled dataset into a Decision Tree model. This time, the model will be trained on the entire dataset as advised in class before predicting with the new data.

In [35]:

```python
# Just like before, all the 28 * 28 images are flattened.
task2_images_flattened = task2_images.reshape(5, 28 * 28)

# Fit the scaled data into a new Decision Tree model.
new_DecisionTree_model = DecisionTreeClassifier(max_depth = 12)
new_DecisionTree_model.fit(X_whole, y_whole)
```

Out[35]:

```
DecisionTreeClassifier(max_depth=12)
```

In [36]:

```python
# Stores the predicted model here for visualization. Will be used as an index for the class_names.
y_test_hat = DecisionTree_model.predict(task2_images_flattened)

fig, ax = plt.subplots(1, 5, figsize=(15, 15))
for i, axi in enumerate(ax.flat):
    axi.imshow(task2_images[i].astype('uint8'), cmap='gray')
    # The titles will designate how the model classified the images.
    axi.set_title(class_names[y_test_hat[i]])
    axi.set(xticks=[], yticks=[])
```

In [37]:

```python
in_sample_acc = accuracy_score(y_whole, new_DecisionTree_model.predict(X_whole), normalize = True) * 100
out_of_sample_acc = accuracy_score(np.array([0, 2, 9, 7, 1]), y_test_hat, normalize = True) * 100
print("In-sample Accuracy: ", in_sample_acc)
print("Out-of-sample Accuracy: ", out_of_sample_acc)
```

```
In-sample Accuracy:  88.21285714285715
Out-of-sample Accuracy:  0.0
```

None of the images were classified accurately with our Decision Tree model. Here, it was thought that maybe if I went and used my second choice (the Random Forest classifier), it would yield more correct results. Below is the attempt:

In [38]:

```python
# Stores the RandomForestClassifier with the optimal max_depth from Section 5a.
new_RandomForest_model = RandomForestClassifier(n_estimators = 100, max_depth = 15, random_state = 0)
new_RandomForest_model.fit(X_whole, y_whole)

# Stores the predicted model that will be used as an index for displaying purposes.
y_test_hat = new_RandomForest_model.predict(task2_images_flattened)

fig, ax = plt.subplots(1, 5, figsize=(15, 15))
for i, axi in enumerate(ax.flat):
    axi.imshow(task2_images[i].astype('uint8'), cmap='gray')
    # The set title labels will indicate what the predictive model has classified the images.
    axi.set_title(class_names[y_test_hat[i]])
    axi.set(xticks=[], yticks=[])
```

In [39]:

```python
# Stores the accuracy scores for the Random Forest model.
in_sample_acc = accuracy_score(y_whole, new_RandomForest_model.predict(X_whole), normalize = True) * 100
out_of_sample_acc = accuracy_score(np.array([0, 2, 9, 7, 1]), y_test_hat, normalize = True) * 100

# Displays the Accuracy scores.
print("In-sample Accuracy: ", in_sample_acc)
print("Out-of-sample Accuracy: ", out_of_sample_acc)
```

```
In-sample Accuracy:  95.99571428571429
Out-of-sample Accuracy:  0.0
```

None of the images are predicted accurately. After careful consideration, it was theorized that the images were not properly Grayscaled in the same fashion for many MNIST datasets. The images used do not possess the same perfectly blank and white backgrounds like in the provided Fashion dataset.

Below is the execution of rescaling and grayscaling the images to make it similar to the MNIST dataset.

In [40]:

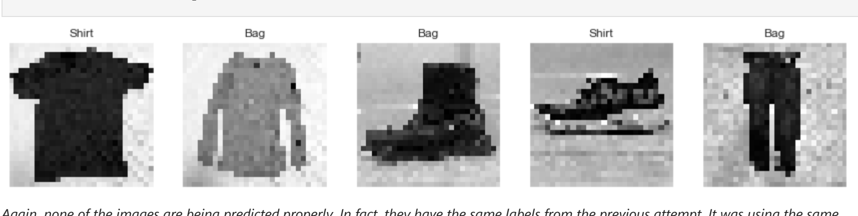
```python
import sys
!{sys.executable} -m pip install opencv-python
import cv2

# Image of t-shirt/top.
task2_image_1 = r'Task 2/IMG_0956.jpg'
task2_image_1 = cv2.imread(task2_image_1, cv2.IMREAD_GRAYSCALE)
task2_image_1 = cv2.resize(task2_image_1, (28, 28), interpolation = cv2.INTER_LINEAR)

# Image of pullover.
task2_image_2 = r'Task 2/IMG_0957.jpg'
task2_image_2 = cv2.imread(task2_image_2, cv2.IMREAD_GRAYSCALE)
task2_image_2 = cv2.resize(task2_image_2, (28, 28), interpolation = cv2.INTER_LINEAR)

# Image of ankle boot.
task2_image_3 = r'Task 2/IMG_0958.jpg'
task2_image_3 = cv2.imread(task2_image_3, cv2.IMREAD_GRAYSCALE)
task2_image_3 = cv2.resize(task2_image_3, (28, 28), interpolation = cv2.INTER_LINEAR)

# Image of sneaker.
task2_image_4 = r'Task 2/IMG_0959.jpg'
task2_image_4 = cv2.imread(task2_image_4, cv2.IMREAD_GRAYSCALE)
task2_image_4 = cv2.resize(task2_image_4, (28, 28), interpolation = cv2.INTER_LINEAR)

# # Image of trousers.
task2_image_5 = r'Task 2/IMG_0960.jpg'
task2_image_5 = cv2.imread(task2_image_5, cv2.IMREAD_GRAYSCALE)
task2_image_5 = cv2.resize(task2_image_5, (28, 28), interpolation = cv2.INTER_LINEAR)

# Array to store and load the images on demand.
task2_images = np.array([task2_image_1, task2_image_2, task2_image_3, task2_image_4, task2_image_5])

fig, ax = plt.subplots(1, 5, figsize=(15, 15))
for i, axi in enumerate(ax.flat):
    axi.imshow(task2_images[i].astype('uint8'), cmap='gray')
    axi.set(xticks=[], yticks=[])
```

```
Requirement already satisfied: opencv-python in c:\users\chris\anaconda3\lib\site-packages (4.5.5.64)
Requirement already satisfied: numpy>=1.17.3 in c:\users\chris\anaconda3\lib\site-packages (from opencv-python)
(1.20.3)
```

Now, to try using the model again with these images.

In [41]:

```python
# Just like before, all the 28 * 28 images are flattened.
task2_images_flattened = task2_images.reshape(5, 28 * 28)

# Fit the scaled data into a new Decision Tree model.
new_DecisionTree_model = DecisionTreeClassifier(max_depth = 12)
new_DecisionTree_model.fit(X_whole, y_whole)

y_test_hat = DecisionTree_model.predict(task2_images_flattened)

fig, ax = plt.subplots(1, 5, figsize=(15, 15))
for i, axi in enumerate(ax.flat):
    axi.imshow(task2_images[i].astype('uint8'), cmap='gray')
    axi.set_title(class_names[y_test_hat[i]])
    axi.set(xticks=[], yticks=[])
```



Again, none of the images are being predicted properly. In fact, they have the same labels from the previous attempt. It was using the same model that was fitted the same way.

However, looking back at the Precision-Recall metrics in the previous sections, some class labels like Shirt were left to be desired. For some innovation, I have taken the liberty of thinking that if five more images with the class labels of higher/more favorable overall Precision/Recall metrics like Bag and Sandals were used, the model would properly identify them.

In [42]:

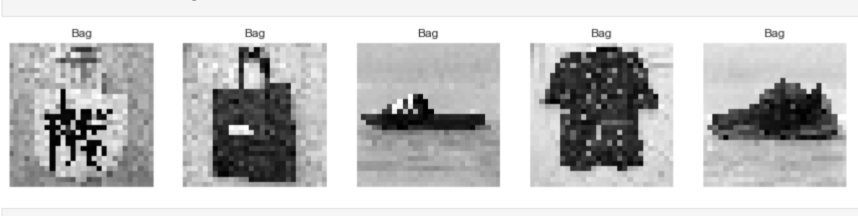
```python
# Defining my images as individual arrays. Each of the images have the shape of (3024, 3024, 3).
task2_image_1 = np.array(mpimg.imread('Task 2 Extra/image0.jpeg')) # T-shirt/top
task2_image_2 = np.array(mpimg.imread('Task 2 Extra/image1.jpeg')) # Pullover
task2_image_3 = np.array(mpimg.imread('Task 2 Extra/image2.jpeg')) # Sneaker
task2_image_4 = np.array(mpimg.imread('Task 2 Extra/image3.jpeg')) # Ankle boot
task2_image_5 = np.array(mpimg.imread('Task 2 Extra/image4.jpeg')) # Trousers

# Loading the individual arrays into a single collective array. An array of 5 images.
task2_images = np.array([task2_image_1, task2_image_2, task2_image_3, task2_image_4, task2_image_5])

# An array used to index the class_names for the intended labels of the images.
intended = np.array([8, 8, 5, 6, 7])

# Visualizing the images side by side.
fig, ax = plt.subplots(1, 5, figsize = (15, 15))
for i, axi in enumerate(ax.flat):
    axi.imshow(task2_images[i])
    axi.set_title(class_names[intended[i]])
    axi.set(xticks=[], yticks=[])

# Displaying the shape of the new images. Proof that they're consistent with the earlier attempt.
print("Shape for each of the individual images: ", task2_image_1.shape)
print("Shape of task2_images array: ", task2_images.shape)
```

```
Shape for each of the individual images:  (3024, 3024, 3)
Shape of task2_images array:  (5, 3024, 3024, 3)
```

As we can see here, these are all new images with their corresponding classes as the title. Images will be now rescaled and grayscaled in a similar manner to that of the MNIST dataset.

In [43]:

```python
# Image of a bag.
task2_image_1 = r'Task 2 Extra/image0.jpeg'
task2_image_1 = cv2.imread(task2_image_1, cv2.IMREAD_GRAYSCALE)
task2_image_1 = cv2.resize(task2_image_1, (28, 28), interpolation = cv2.INTER_LINEAR)
#task2_image_1 = cv2.bitwise_not(task2_image_1)

task2_image_2 = r'Task 2 Extra/image1.jpeg'
task2_image_2 = cv2.imread(task2_image_2, cv2.IMREAD_GRAYSCALE)
task2_image_2 = cv2.resize(task2_image_2, (28, 28), interpolation = cv2.INTER_LINEAR)
#task2_image_2 = cv2.bitwise_not(task2_image_2)

task2_image_3 = r'Task 2 Extra/image2.jpeg'
task2_image_3 = cv2.imread(task2_image_3, cv2.IMREAD_GRAYSCALE)
task2_image_3 = cv2.resize(task2_image_3, (28, 28), interpolation = cv2.INTER_LINEAR)
#task2_image_3 = cv2.bitwise_not(task2_image_3)

task2_image_4 = r'Task 2 Extra/image3.jpeg'
task2_image_4 = cv2.imread(task2_image_4, cv2.IMREAD_GRAYSCALE)
task2_image_4 = cv2.resize(task2_image_4, (28, 28), interpolation = cv2.INTER_LINEAR)
#task2_image_4 = cv2.bitwise_not(task2_image_4)

task2_image_5 = r'Task 2 Extra/image4.jpeg'
task2_image_5 = cv2.imread(task2_image_5, cv2.IMREAD_GRAYSCALE)
task2_image_5 = cv2.resize(task2_image_5, (28, 28), interpolation = cv2.INTER_LINEAR)
#task2_image_5 = cv2.bitwise_not(task2_image_5)

task2_images = np.array([task2_image_1, task2_image_2, task2_image_3, task2_image_4, task2_image_5])

# Just like before, all the 28 * 28 images are flattened.
task2_images_flattened = task2_images.reshape(5, 28 * 28)

# Fit the scaled data into a new Decision Tree model.
new_DecisionTree_model = DecisionTreeClassifier(max_depth = 12)
new_DecisionTree_model.fit(X_whole, y_whole)

y_test_hat = DecisionTree_model.predict(task2_images_flattened)

fig, ax = plt.subplots(1, 5, figsize=(15, 15))
for i, axi in enumerate(ax.flat):
    axi.imshow(task2_images[i].astype('uint8'), cmap='gray')
    axi.set_title(class_names[y_test_hat[i]])
    axi.set(xticks=[], yticks=[])
```



In [44]:

```python
# An array used to index the class_names for the intended labels of the images.
intended = np.array([8, 8, 5, 6, 7])

# Stores the accuracy scores for the Random Forest model.
in_sample_acc = accuracy_score(y_whole, new_DecisionTree_model.predict(X_whole), normalize = True) * 100
out_of_sample_acc = accuracy_score(intended, y_test_hat, normalize = True) * 100

# Displays the Accuracy scores.
print("In-sample Accuracy: ", in_sample_acc)
print("Out-of-sample Accuracy: ", out_of_sample_acc)
```

```
In-sample Accuracy:  88.21142857142857
Out-of-sample Accuracy:  40.0
```

This time, the model was able to predict two out of five of the new provided images. This is certainly an improvement from the previous attempts. Still, it's not predicting as favorably as anticipated from Task 1. As hard as I tried, these new images still have a not entirely blank background. I strongly believe this is distorting the model from making accurate predictions.

Output

- *Make sure to put descriptive comments on your code*
- *Use the markdown cell format in Jupiter to add your own interpretation to the result in each section.*
- *Make sure to keep the output of your runs when you want to save the final version of the file.*
- *The final work should be very well structured and should have a consistent flow of analysis.*

Due Date: Apr 5 2022 at 7:00 PM

Grading Criteria

Comprehensiveness	***30%***
Correctness	***20%***
Complete Report	***20%***
Clear Code	***20%***
Innovation (Extra)	***20%***
Total	***110%***